VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

METODY PRO ZOBRAZENÍ MĚKKÝCH STÍNŮ

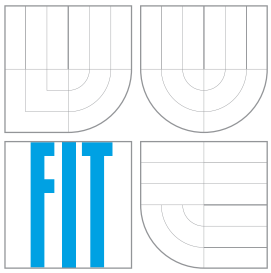
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ ONDRUŠKA

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

METODY PRO ZOBRAZENÍ MĚKKÝCH STÍNŮ

METHODS FOR SOFT SHADOWS RENDERING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JIŘÍ ONDRUŠKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN NAVRÁTIL
SUPERVISOR

BRNO 2012

Abstrakt

Práce probírá dva rozdílné přístupy ke tvorbě měkkých stínů. Metodu stínových těles a metodu stínových map, přesněji variance soft shadow mapping. Práce uvádí teorii nezbytnou k pochopení jednotlivých algoritmů. Následuje popis a zhodnocení implementace těchto metod. Metoda stínových těles pracuje na principu sestavování přídavných geometrií do scény sloužících k určení prostoru polostínu. VSSM algoritmus je pak vylepšením klasického algoritmu stínových map.

Abstract

This thesis discusses two different methods for creating soft shadows. Shadow volumes and shadow mapping, more accurately Variance Soft Shadow Mapping. It presents theory for these shadow algorithms as well as theory for few others which are necessary for understanding. Further it describes how to implement these methods and evaluates these implementations. Shadow volumes are based on creating additional geometry to the scene which serves for specifying region of penumbra. VSSM algorithm is a improved version of classic shadow mapping.

Klíčová slova

měkké stíny, stínová tělesa, stínové mapy, zobrazení v reálném čase, Čebyševova nerovnost, integrální obraz, hierarchická stínová mapa, OpenGL, GLSL

Keywords

soft shadows, shadow volumes, shadow maps, realtime rendering, Chebychev's inequality, summed area table, hierarchic shadow map, OpenGL, GLSL

Citace

Jiří Ondruška: Metody pro zobrazení měkkých stínů, diplomová práce, Brno, FIT VUT v Brně, 2012

Metody pro zobrazení měkkých stínů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Jana Navrátila

.....
Jiří Ondruška
22. května 2012

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce, panu Janu Navrátilovi, za motivující konzultace a poskytnuté rady a mým rodičům za poskytnutou podporu během psaní této diplomové práce.

Obsah

Kapitola 1

Úvod

Hra světla a stínu. Meka dnešní počítačové grafiky. Stíny nejen přidávají na realistickém vzhledu výsledného snímku či scény, ale také pomáhají pozorovateli lépe určit vzájemné pozice mezi objekty v prostoru. Existuje však spousta způsobů jak lze stíny získat. V následujících stranách proberu odlišné přístupy ke tvorbě stínů. Přesněji ke tvorbě měkkých stínů, které jsou pro realistický vzhled scény nezbytné, jelikož většina zdrojů ve světě je plošných.

Nejen v astronomii, ale i v počítačové grafice se stín dělí na tři části. První částí je nejtemnější oblast stínu, tzv. umbra (latinsky „stín“, z řeckého „ombros“). Druhou částí je penumbra, oblast plynulého přechodu mezi plným stínem a plným světlem. Sestavení oblasti penumbry co nejvěrohodněji, ale s pokud možno co nejmenší náročností na výpočetní výkon, je hlavním tématem metod pro zpracování měkkých stínů. Poslední částí stínu, zajímavou převážně z astronomického hlediska je antumbra, avšak i v počítačové grafice se využití určitě najde. Antumbra je oblast z níž je vidět prstencové zatmění, vznikající pokud je zdroj světla větší než objekt, který jej překrývá. Oblast pozorování úplného zatmění slunce je dobrým příkladem.

Nárust výkonu v oblasti procesů a grafických čipů znamená, že je možné prakticky zrealizovat algoritmy, které v době uvedení, neměly potenciál díky své náročnosti. Výkon dnešního hardware dává prostor robustním algoritmům, které se snaží o co největší realismus. Stínové algoritmy lze rozčlenit do tří kategorií: stínová tělesa, stínové mapy a projekční stíny. Každá kategorie má své výhody a nevýhody. Tato práce se zaměří na zástupce prvních dvou, tedy stínových těles a stínových map.

Nejdříve uvedu teorii k algoritmu stínových těles, následně popíši teorii ke stínovým mapám. Cílem této práce je zhodnotit jednotlivé přístupy ke tvorbě stínů, jejich výhody a nevýhody, též zhodnotit náročnost těchto algoritmů na pochopení a náročnost na implementaci.

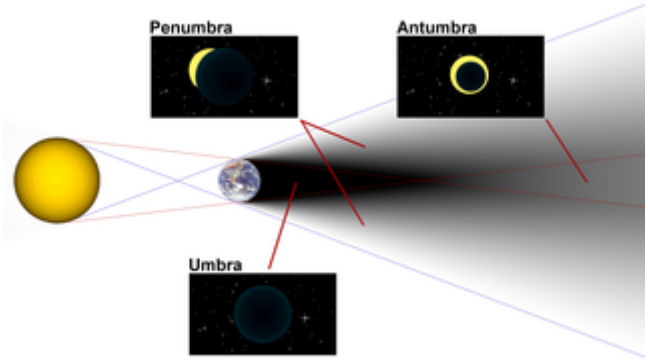
Kapitola 2

Měkké stíny

Vytváření měkkých stínů (angl. *soft shadows*) je základním a podstatně náročným problémem v počítačové grafice. Obecně stíny nejen přidávají na realistickém vzhledu výsledného snímku či scény, ale také pomáhají pozorovateli lépe určit vzájemné pozice mezi objekty v prostoru. Ve skutečném světě jsou nejčastější právě měkké stíny, jelikož většina světelných zdrojů je plošná.

Měkký stín lze rozložit na dvě hlavní části. *Plný stín* (angl. *umbra*: zastíněná oblast), tedy oblast kam se ze zdroje nedostává žádné světlo, *polostín* (angl. *penumbra*), což je oblast hladkého přechodu mezi plným stínem (žadným světlem) a plným světlem. Třetí méně zajímavou částí je antumbra, oblast, ze které lze pozorovat prstencové zatmění. Jednotlivé oblasti stínu lze pěkně pozorovat na jednoduchém modelu Slunce-Země, viz obrázek 2.1.

Bodové zdroje světla negenerují polostín, ale jen plný stín. Tento efekt se běžně nazývá tvrdým stínem (angl. *hard shadow*). Bodové zdroje světla se v reálném světě objevují výjímečně. Navíc tvrdý stín může být pozorovatelem chybně vnímán jako geometrická vlastnost, což určitě není vhodné. Pro tyto důvody se obecně preferuje využití měkkých stínu, to ovšem neznamená, že by tvrdé stíny neměly využití, např. jako mezikrok k měkkým stínům.



Obrázek 2.1: Ukázka oblastí stínu.

2.1 Stínová tělesa

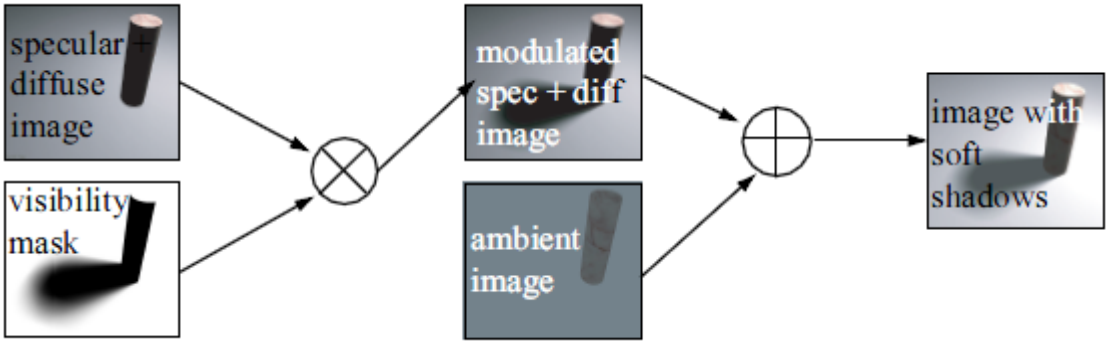
Shadow volumes je technika používaná v počítačové grafice pro vytváření stínů ve scéně. Poprvé se objevili v roce 1977 [7] jako geometrie popisující oblast ve trojrozměrném pro-

storu, která je v zákrytu vůči světlu. Stínové těleso rozděluje prostor virtuálního prostředí na dvě části: oblasti, které jsou ve stínu, a oblasti mimo stín.

Stínová tělesa se stala poměrně populárními pro realtime aplikace. Hlavní výhodou je jejich přesnost (na pixel), avšak mnoho implementací má menší problémy se sebezastíněním (angl. *self-shadowing*) podél hran siluety. Za přesnost se však platí nutností sestavení přídavné geometrie, což může být náročné na procesor. V dnešní době, kdy lze geometrii měnit pomocí geometry shaderu na grafickém jádře, lze tuto nevýhodu relativně slušně redukovat.

2.1.1 Popis algoritmu

Pro potřeby tohoto algoritmu [4] je nutné prvně scénu vyrenderovat s použitím dvou složek světla – *spekulární* (angl. *specular*: zrcadlová), *difúzní* (angl. *diffuse*: rozptýlená). Po-té se spočítá maska viditelnosti, která se použije k úpravě již vyrenderované obrazu z první části. Následně se vyrenderuje scéna s *ambientní* (angl. *ambient*: okolní) složkou světla. Jako finální průchod se pak tyto dvě části spojí. Tento proces ilustruje obrázek 2.2.



Obrázek 2.2: Grafické znázornění procesu tvorby měkkých stínů z jednotlivých složek. Algoritmus se zabývá vytvořením masky viditelnosti - vlevo dole. [4]

Maska viditelnosti neboli V-buffer uchovává faktor viditelnosti $\bar{v}$ pro každý pixel(x, y) obrazu. Pokud bod $p = (x, y, z)$, kde z je hodnota z hloubkové mapy scény (angl. *depth map*) neboli hodnota Z-bufferu na pozici (x, y), „vidí“ všechny body zdroje světla tj. bez zakrytí, pak $\bar{v}$ je 1. Protikladem je pak plně zakrytý bod a má tedy faktor viditelnosti $\bar{v}$ roven 0. Bod, který „vidí“ x procent zdroje světla, má tedy $\bar{v} = x/100 \in (0, 1)$. Tedy má-li bod $0 < \bar{v} < 1$, pak je tento bod v oblasti polostínu.

2.1.2 Spočtení viditelnosti

Výpočet viditelnosti je rozdělen na dvě části (průchody). První průchod (2.1.4) spočítá tvrdé stíny, např. algoritmem [7]. Tyto tvrdé stíny jsou uloženy do V-bufferu, čímž se nadhodnotí oblast plného stínu. Takto získaná maska jde využít i pro zjistění vstupu do/výstupu ze stínu.

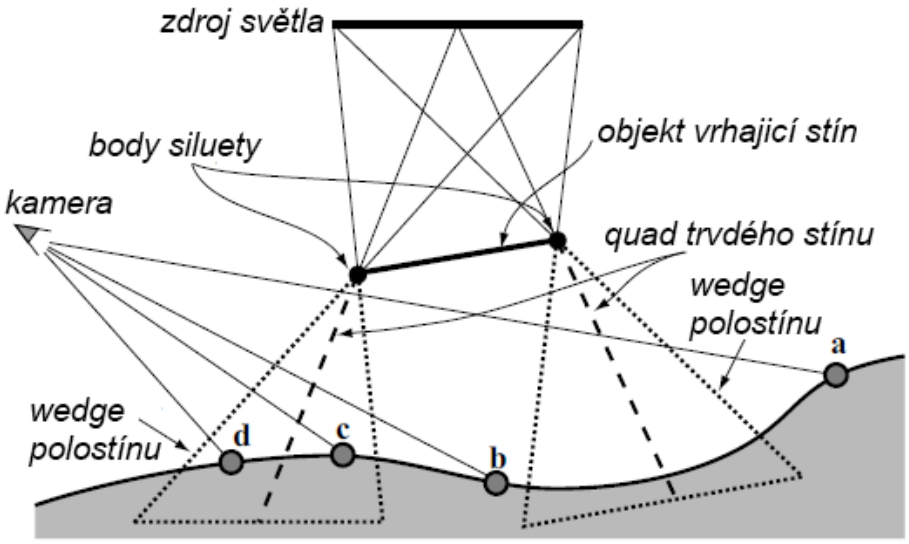
Druhý průchod (2.1.5) pak renderuje výseče (angl. *wedge*: klín, výseč) polostínu (2.1.3), aby kompenzovaly přehodnocený stín z prvního průchodu. Dohromady tak vzniknou měkké stíny.

Obrázek 2.3 ukazuje na dvou rozměrech jak tyto průchody spolupracují. Viditelnost pro body **a**, **b**, **c** a **d** se spočtou následovně. Pro body **a** a **b** algoritmus funguje stejně jako algoritmus pro tvrdé stíny, jelikož oba tyto body leží mimo plochy polostínu.

Pro bod **a** jsou vykresleny oba quady (angl. *quad*: čtyřúhelník) tvořící tvrdý stín, díky tomu, že levý je natočený na pozorovatele a pravý od pozorovatele, bod **a** bude mimo stín. Bod **b** leží pouze za levým quadem a je tedy v plném stínu.

Pro bod **c** je vykreslen levý quad a následně během rasterizace wedge je zjištěno, že se bod nachází uvnitř wedge a tedy je v polostínu. Bod je tedy promítnut do světla, aby se zjistil skutečný faktor viditelnosti. Bod **d** je přede všemi quady, ale je uvnitř levé wedge. Je tedy též promítnut do světla a je spočítán jeho skutečný faktor viditelnosti.

Takto získaný faktor je zapsán do V-bufferu.



Obrázek 2.3: Dvourozměrná ukázka spolupráce obou průchodů pro zjistění viditelnosti. Dvě wedge polostínu, generované dvěma body siluety jsou zobrazeny čerchovaně.

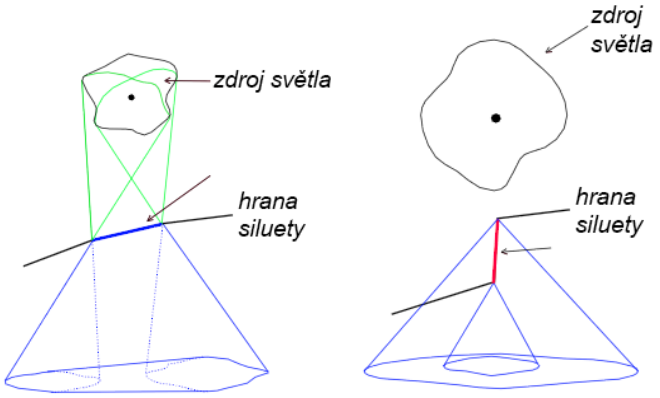
2.1.3 Sestavení wedge pro polostín

Jednou z aproximací v tomto algoritmu je využití pouze hran tvořících siluetu objektu vrhajícího stín, tak jak je viděn z jednoho bodu, nejčastěji středu světelného zdroje. Tato aproximace se používá i v jiných algoritmech [1]. Hrany tvořící siluetu spojují dva trojúhelníky: první je natočený na světlo a druhý je odvracený od světla. Takovouto siluetu lze najít pomocí tzv. hrubé síly, algoritmu, který projde hrany všech trojúhelníků. Lze též použít i sofistikovanější metody ([15]). Avšak vyhledávání siluety je málokdy úzkým hrdlem výpočtů, s vyjímkou modelů s vysokou hustotou polygonů.

Pro obecný zdroj světla je přesný prostor polostínu generován posunem obecného kuželu po hraně siluety od jednoho vrcholu hrany ke druhému. Obecný kužel má pak tvar odpovídající tvaru světla promítáného skrz posunový bod na hraně. Toto lze vidět na obrázku 2.4.

Počítat přesný objem polostínu není příliš vhodné pro aplikace s proměnným prostředím, které by měly běžet v reálném čase. Naštěstí přesnosti není potřeba. V oddílu (2.1.5) věnovanému rasterizaci wedge je popsáno, jak lze výpočty viditelnosti bodů uvnitř jedné wedge rozdělit tak, aby byly nezávislé na ostatních. Postačuje sestavit pouze „obálku“, která plně uzavírá přesný objem polostínu. Například pomocí wedge sestavené ze čtyř rovin (přední, zadní, pravá, levá), jak lze vidět na obrázku 2.5d.

Efektivní spočtení wedge uzavírající přesný objem polostínu vypadá následovně. Hrana siluet je definována dvěma vrcholy, e_0 a e_1. Nejdříve se najde vrchol hrany, který je blíže



(a) (b)

Obrázek 2.4: a) Polostín generovaný hranou. b) Polostín může degradovat do jediného kužele, pokud kužel jednoho koncového bodu plně obklopí druhý.

ke zdroji světla. Beze ztráty na obecnosti lze předpokládat, že e_1 je blíž. Druhý vrchol hrany je pak posunut podél směru do středu světla tak, aby byl stejně daleko jako e_1. Takto přesunutý vrchol se označí e'_0. Tyto dva vrcholy pak vytvoří novou hranu, která bude sloužit jako vršek sestavované wedge (2.7d). Sestavením této nové hrany se zajistí, že wedge bude obsahovat celý objem polostínu původní hrany, ovšem pro výpočty viditelnosti jako takové se použije původní nemodifikované hrany. Jak bude patrné později, tak body uvnitř wedge, které nepatří do skutečného polostínu, vůbec viditelnost neovlivní.

Dále se definují přední a zadní roviny tak, že obsahují novou hranu e'_0e_1 a tyto roviny se rotací kolem této hrany natočí tak, že se sotva dotýkají zdroje světla na každé straně. Pro světelné plochy geometrických tvarů či ploch uzavřených do obálek geometrických tvarů, lze relativně snadno dopočítat přesnou hodnotu. Výsledek lze vidět na obrázku 2.5d.

Pravá rovina je pak definována tak, že obsahuje e'_0 a vektor, který je kolmý jak na hranu e'_0e_1, tak i na vektor z e'_0 do středu světla. Levá rovina je definována podobně, ale z opačné strany. Navíc obě tyto roviny by se měly sotva dotknout světla na každé straně.

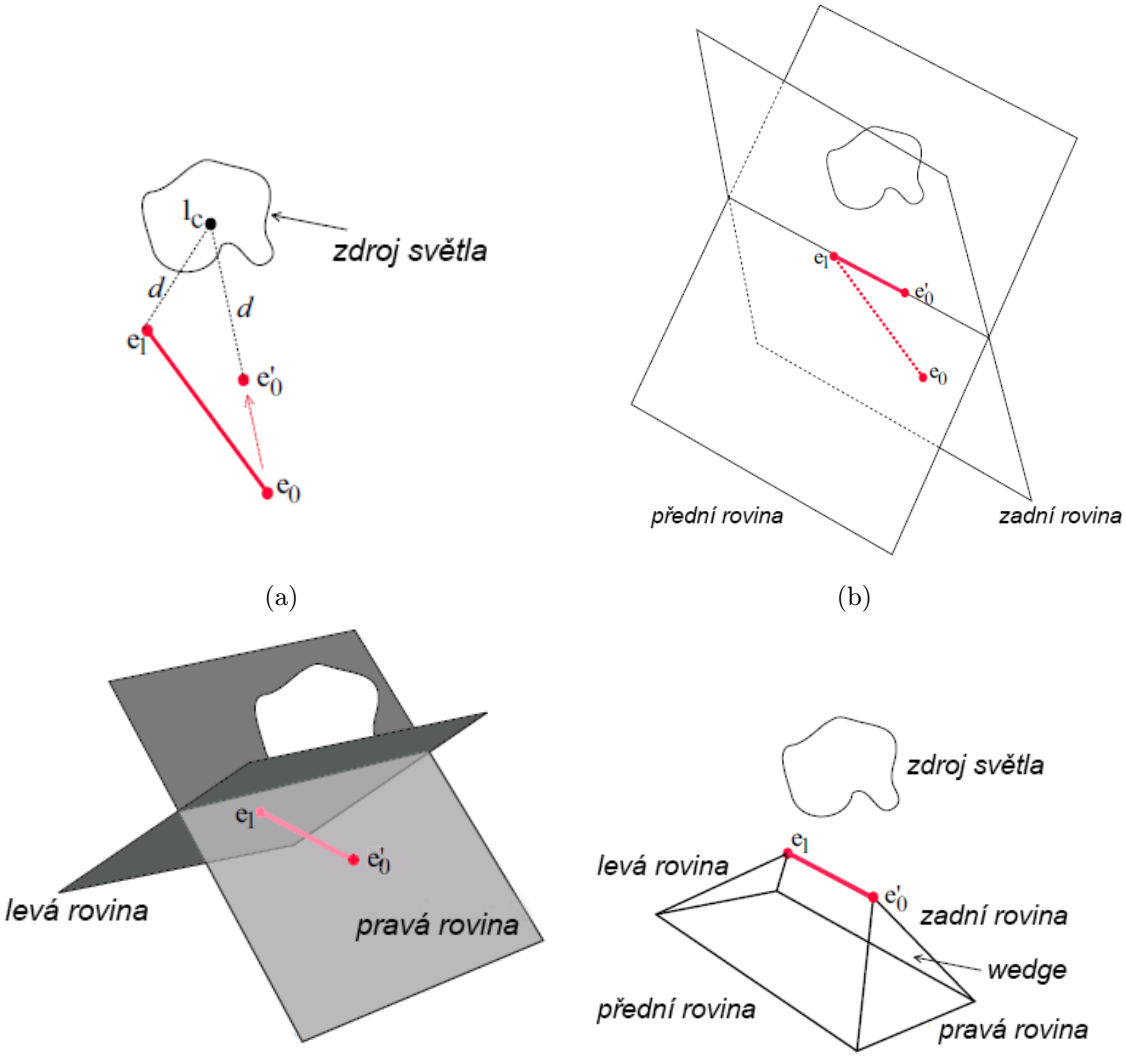
Po sestavení těchto rovin se vyjmou polygony tvořící výslednou wedge. Tyto polygony budou pak použity pro rasterizaci jak je popsáno v oddílu 2.1.5.

Výhodou tohoto přístupu je hlavně to, že wedge jsou sestaveny nezávisle na sobě, což je klíčové pro jednoduchost a rychlost algoritmu. Za zmínku stojí, že pokud vrcholy tvořící hranu siluety jsou ve velmi odlišných vzdálenostech od zdroje světla, obálka nebude příliš těsná. Toto však nemá vliv na výslednou správnost zobrazení, ale má to za následek nadbytečné množství pixelů, které je nutné zpracovat. Na druhou stranu vyřazení pixelu ze zpracování není příliš náročné a při porovnání s časem, který je potřeba pro jeho zpracování, je toto zanedbatelné. Předpokládá se, že objekty nijak neprotínají zdroj světla, avšak mohou světlo plně obklopovat.

2.1.4 Průchod první – plný stín

Během tohoto průchodu se vytvoří tvrdé stíny. Na začátku se V-buffer nastaví celý na hodnotu jedna, což znamená, že vše je mimo stín. Vyrenderují se quady pro tvrdé stíny dle některého z algoritmů. Například lze odečítat jedničku z V-bufferu pro quady natočené k pozorovateli a přičítat jedničku pro natočené od pozorovatele.

Velmi důležitou vlastností při používání quadů tvrdých stínů je to, že nemusíme znát



Obrázek 2.5: Kroky sestavení wedge. a) Posunutí vrcholu vzdáleného více od středu světla l_c směrem k l_c na stejnou vzálenost jako druhý vrchol hrany. b) Vytvoření přední a zadní roviny. c) Vytvoření levé a pravé roviny. d) Výsledná wedge je objem uvnitř přední, zadní, levé a pravé roviny.

přesný povrch mezi polostínem a stínem. Počítání této plochy ve třech rozměrech je složité a časově náročné. Právě díky renderování tvrdých stínů pomocí quadů výpočet této plochy není vůbec potřeba. Na druhou stranu přibude nutnost kompenzovat přesah plného stínu, čehož se dosáhne v druhém průchodu vykreslením jednotlivých výsečí pro polostín.

Je důležité zmínit, že tento první průchod je nezbytný a další zpracování bez něj nedokáže správně určit, zda je bod ve stínu nebo mimo stín. Při vynechání této části lze v dalším zpracování určit jen zda je bod v polostínu.

2.1.5 Průchod druhý – polostín

Cílem tohoto průchodu je kompenzovat přehnanou plochu stínu z první části algoritmu a dopočítat skutečný faktor viditelnosti všech bodů $p = (x, y, z)$ polostínu, kde z je hodnota hloubky ze Z-bufferu na pixelu (x, y), uvnitř každé wedge (výseče). Dále se předpokládá, že je použit čtvercový (či obdélníkový) zdroj světla L, a že stíny v prvním průchodu byly vytvořeny pomocí quadů sestavených s pomocí středu toho čtvercového světla.

Pro spočítání viditelnosti bodu p, dle sady hran tvořící siluetu objektu vrhajícího stín, si lze představit, že pozorovatel se nachází v bodě p a dívá se na zdroj světla L. Viditelnost bodu p je potom úměrná ploše světelného zdroje, kterou pozorovatel vidí, vydělená celkovou velikostí plochy světla. Více se o zpětné projekci lze dočíst například v článku [11].

Soustřeďme se dále jen na jednu wedge polostínu generovanou právě jednou hranou siluety, e_0e_1, a na bod p ležící uvnitř této wedge. V následujících řádcích popíšu jak spočítat faktor viditelnosti pro bod p s ohledem na hranu e_0e_1, dále pak bude následovat vysvětlení jak soubor viditelností všech výsečí udává celkový vzhled měkkého stínu.

Nejdříve se quad Q, tvořící tvrdý stín, promítne na zdroj světla skrz hranu e_0e_1, jak je viděn z bodu p. Tato projekce se skládá z promítnuté hrany, nekonečných hran vedoucích z jejích koncových bodů, paralelních s vektorem ze středu světla do promítnutých koncových bodů hrany. Toto lze vidět na obrázku 2.6.

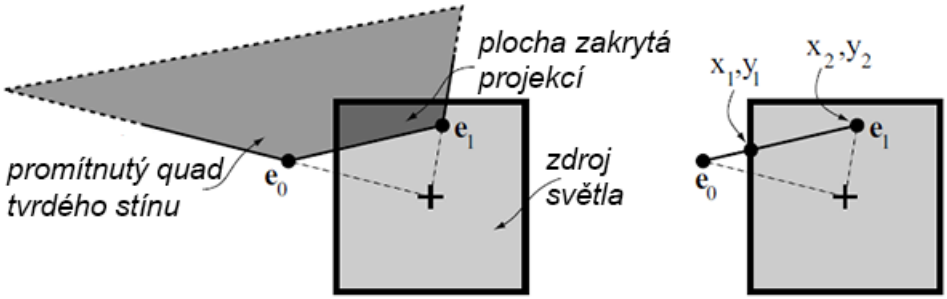
Následně se spočítá plocha průniků mezi světlem a promítnutým quadem, a následně je vydělena celkovou obsahem plochy světla. Toto se nazývá pokrytí světla a na obrázku 2.6 jde o tmavě šedou barvu. Pro přesné výpočty lze použít jednoduchý algoritmus pro ořezávání.

Algoritmus 1 Pseudokód rasterizace wedge

```
1:  function RASTERIZEWEDGE(wedge W, hard_shadow_quad Q, light L)
2:      pro každý pixel(x, y) náležící trojúhelníku wedge natočenému na pozorovatele:
3:      p = (x, y, z) // z je hloubka ze Z-bufferu
4:      if p je uvnitř wedge then
5:          v_p = promitnoutQuadASpocistPokryti(W,p,Q)
6:          if p je v kladném poloprostoru Q then
7:              v̄(x, y) = v̄(x, y) − v_p
8:          else
9:              v̄(x, y) = v̄(x, y) + v_p
10:         end if
11:     end if
12:     return v̄
13: end function
```

Pseudokód 1 pro rasterizaci wedge je pak relativně jednoduchý. Pokud je tento kód použit pro všechny siluety, viditelnou plochu světla lze v podstatě spočítat pomocí Greenova teorému [14]. Pokud je řádek číslo 4 pravdivý, tak p může být v oblasti polostínu a je zapotřebí dalších výpočtů. Řádek 5 promítne quad na světlo a spočítá kolik procent světla je z daného bodu překryto quadem. Funkce může překrytí přímo spočítat a nebo načíst z předpočítaných hodnot, více v popisu implementace (kapitola 3). Plocha překrytí odpovídá tmavé ploše na obrázku 2.6. Rovina quadu Q rozděluje prostor na dva poloprostory záporný a kladný. Záporný poloprostor je definován tak, že je částí plného stínu. Tato informace je potřeba na řádku 6, kde se tak určí, která operace (sčítání/odečítání) se má použít pro vyhodnocení Greenova teorému. Příklad toho, jak toto funguje lze vidět na obrázku 2.7,

který zobrazuje pohled z bodu p směrem do světla. Šedá oblast je stínítko, tedy objekt vrhající stín, viděný z bodu p. Hrany A a B přispívají k celkové viditelnosti z p. Nastavením příspěvků z A a B tak, že odpovídají virtuálním stínítkům, jak je vidět na obrázku 2.7, lze viditelnou plochu spočíst bez znalostí celé siluety.



Obrázek 2.6: Vlevo: Výpočet zakrytí světla pro bod p dle hrany e_0e_1. Zobrazen je pohled z bodu na střed světla. Toto je možno si představit jako projekci quadu Q tvrdého stínu na zdroj světla tak, jak je viděn z bodu p. Vpravo: Hrana je oříznuta okrajem zdroje světla. Toto dáva vzniknout čtveřici (x_1, y_1, x_2, y_2) sloužící jako index do čtyřrozměrné vyhledávací tabulky.

2.2 Stínové mapy

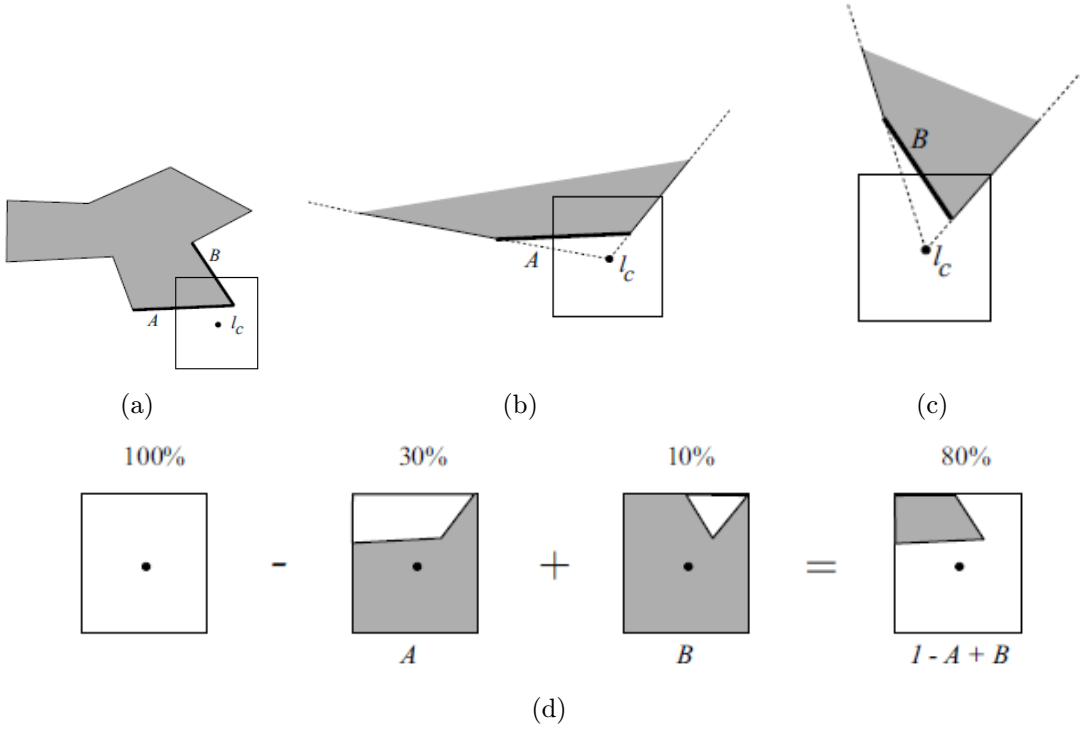
Mapování stínů (angl. *shadow mapping*) je skupina čistě obrazově založených technik. Zpracovává se tedy aktuální snímek a komplexnost scény má zanedbatelný vliv na výkon těchto technik. V dnešní době se z této kategorie de facto stává standard pro počítačové hry. Původně tento typ stínů řešil jen tvrdé stíny. Postupně se objevilo několik základních úprav, které řeší právě měkké stíny. Tyto techniky nejsou nikterak nové, existují od konce sedmdesátých let.

Princip těchto stínu je snadný. Vyrenderuje se scéna z pohledu světla. Při renderování scény samotné se pak testuje porovnáním hloubek, jestli je daný bod ze světla vidět. Jinak řečeno, umístíli se pozorovatel do světla, vše, co vidí, je osvětleno, vše, co je za těmito objekty, je ve stínu.

Tyto techniky jsou v základu méně přesné než techniky založené na stínových tělesech. Výhodou však je větší rychlost zpracování, kterou jen minimálně ovliňuje komplexnost scény, počet polygonů apod. Shadow mapping též nepotřebuje přídavný stencil buffer jako tomu je u stínových těles. Přesnost stínové mapy je limitována pouze jejím rozlišením.

Shadow mapping již díky základům nebude nikdy dokonale přesný, existuje však velké množství technik, které zlepšují výsledný vzhled. Čistý shadow mapping též vytváří jen tvrdé stíny, ale jelikož stíny jsou tvořeny texturou není problém s ní provést různé úpravy, aby se získali měkké stíny.

Existuje řada technik pro měkké stíny, které vychází ze shadow mappingu. V zásadě tyto techniky lze rozdělit do několika kategorií dle přístupu ke stínové mapě: filtrovací, změkčovací, rozdělovací a měkké stíny jako takové. Pro pochopení algoritmu, který jsem vybral k implementaci je třeba znát alespoň základy hlavních zástupců těchto kategorií. Uvedu zde jen to hlavní, co je třeba o těchto algoritmech znát, bližší informace lze získat snadno na webových stránkach nebo v referovaných článcích.



Obrázek 2.7: Rozdělení příspěvků jednotlivých výsečí na celkové viditelnosti pro bod p. A a B jsou hrany siluety objektu vrhajícího stín. [4]

2.2.1 Percentage Closer Filtering

Jak anglický název napovídá, jedná se o filtrování klasické stínové mapy. Na rozdíl od klasických textur nelze stínové mapy předfiltrovat. Místo toho se použije více vzorků ze shadow mapy a tyto se zprůměrují dohromady. Toto je základem PCF [6], spočítá se vlastně kolik procent povrchu je blíže ke světlu a tedy není ve stínu. První verze této techniky vybírala vzorky nahodile. Dnes se používá statický výběr vzorků.

Tato technika je například použita v hardwaru společnoti NVIDIA, kde se provádějí čtyři porovnání hloubky a bilineárně se interpoluje hodnota stínu.

2.2.2 Percentage Closer Soft Shadows

Tato technika je vlastně vylepšenou podobou PCF. Opětovně se bere několik vzorků ze stínové mapy, které slouží k výpočtu průměrné hloubky a k určení stínu. Zásadním rozdílem je však výběr těchto vzorků.

Percentage Closer Soft Shadows [12] je založeno na pozorování PCF, kdy při zvětšování jádra PCF se výsledné stíny stávají jemnější. PCSS pak zajišťuje inteligentní změnu velikosti jádra PCF filtru tak, aby výsledek měl správný stupeň rozmazání.

Algoritmus funguje ve třech krocích. Prvním krokem je tzv. blocker search, tedy hledání stínítka. Projde se stínová mapa a zprůměrují se hloubky, které jsou blíže ke světlu než řešený bod (receiver). Velikost prohledávané plochy se odvodí z velikosti světla a vzdálenosti řešeného bodu od světla.

Druhý krok je odhad šířky polostínu. Použitím aproximace pomocí rovnoběžných rovin, se určí šírka polostínu ($w_{penumbra}$) pomocí velikosti světla (w_{light}), vzdálenosti blockeru od

světla ($d_{blocker}$) a vzdálenosti řešeného bodu od světla ($d_{receiver}$) pomocí rovnice (2.1).

$$w_{penumbra} = (d_{receiver} - d_{blocker} * w_{light}/d_{blocker}) \tag{2.1}$$

Třetím krokem je pak samotné filtrování shodné s PCF za použití jádra o odpovídající velikosti z předchozího kroku.

2.2.3 Variance Shadow Maps

U tohoto algoritmu ([10]) rozepíši trochu více matematiky, jelikož je podstatný k pochopení dalších částí této práce. Stejně jako u běžného shadow mappingu se scéna prvně vykreslí z pohledu světla. Zde však je již první rozdíl. Nerenderuje se pouze hloubka, ale i mocnina hloubky. Při normálních stínových mapách se jakákoliv forma anti-aliasingu nepoužívá, jelikož znehodnocuje stínovou mapu. Zde se naopak dá využít ke zlepšení výsledku.

Když je stínová mapa vyrenderována, lze ji předběžně zpracovat pomocí filtrů. Toto může zahrnovat vygenerování mipmap nebo integrálního obrazu (angl. *summed area table*). Pro další zredukování aliasů a zjemnění hran stínu lze též rozostřit.

Jelikož v textuře je uchována jak hloubka tak i její mocnina, výsledkem filtrování textury bude získaní momentů M_1 a M_2 přes danou oblast filtru, a to následovně:

$$M_1 = E(x) = \int_{\infty}^{\infty} xp(x)dx \tag{2.2}$$

$$M_2 = E(x^2) = \int_{\infty}^{\infty} x^2p(x)dx \tag{2.3}$$

Z tohoto lze odvodit střední hodnotu μ a varianci σ (angl. *variance*: rozptyl, odchylka, variance):

$$\begin{aligned} \mu &= E(x) = M_1 \\ \sigma^2 &= E(x^2) - E(x)^2 = M_2 - M_1^2 \end{aligned} \tag{2.4}$$

Varianci lze chápat jako kvantitativní míru šířky distribuce. Ve výsledku by variance měla udávat hranici toho, jaká část distribuce může být nahromaděna daleko od průměru. Tato mez bude přesně specifikována v Čebyševově nerovnosti.

Teorém Čebyševovy nerovnosti Nechť x je nahodná proměnná z distribuce se střední hodnotou μ a variancí σ^2. Pak pro $t > \sigma$

$$P(x \geq t) \leq p_{max}(t) \equiv \frac{\sigma^2}{\sigma^2 + (t - \mu)^2} \tag{2.5}$$

Množství $P(x \geq t)$ v rovnici (2.5) je přesné množství, které by se mělo spočíst, aby se dosáhlo PCF, jelikož reprezentuje zlomek pixelů přes oblast filtru, který neprojde porovnáním hloubky s fixní hloubkou t.

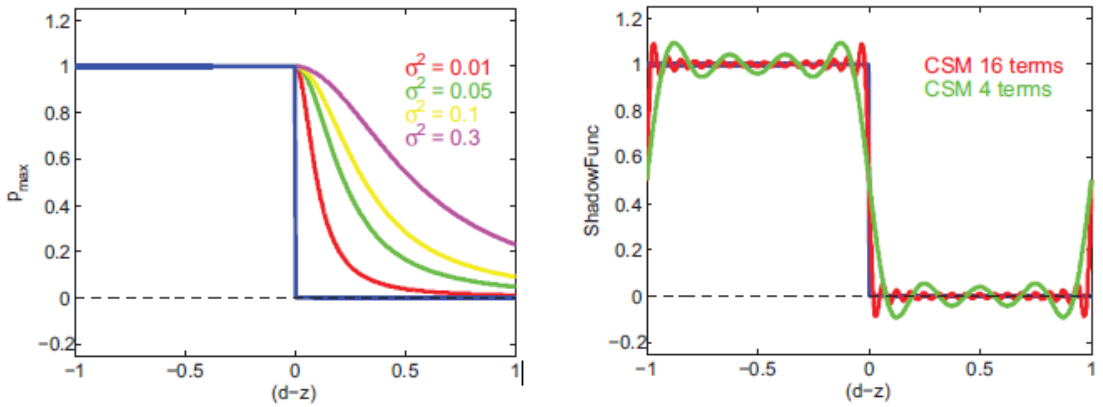
Předchozí rovnice je však horní závorou, proto není důvod předpokládat, že by povolovala spočíst skutečnou hodnotu $P(x \geq t)$, může však poskytnout dobrou aproximaci.

2.2.4 Variance Soft Shadow Mapping

Mnou zvolený algoritmus [16] měkkých stínů spojuje několik různých technik. V prvé řadě jako základ se využívá Variance shadow maps, do čehož se napasuje kostra z Percentage Closer Soft Shadows. Dva dosti odlišné přístupy, které není snadné spojit a jejichž spojení přináší další problémy, které je nutné řešit.

Jedním poučením z PCSS je, že jeho kroky vyžadují získávání vzorků hloubky hrubou silou. Pokud se tedy plošné světlo příliš zvětší je zapotřebí příliš velkého množství vzorků (např. 30x30), aby se zabránilo vzniku artefaktů. Z PCSS vychází efektivnější metody pro předfiltrování např. SAT (angl. *summed area table*: integrální obraz).

Pro použití v PCSS bylo uvedeno několik předfiltrovacích metod. Jednou z těchto technik je například Convolution soft shadow map (CSSM [2]) využívající Fourierovy báze pro rekonstrukci předfiltrovaných stínů. Funkci rekonstrukce s různými bázovými termy lze vidět na obr. 2.8b. Lze pozorovat, že křivka rekonstrukce CSSM pokrývá celý rozsah hodnot $(d - z)$. Takto oboustranně uzavřenou filtrovací funkci lze aplikovat na výpočet průměrné hloubky blockeru i samotného testu měkkých stínů a zapadá do kostry PCSS velice dobře. Avšak tato technika používá velké množství paměti pro uchování Fourierových bází, což není příliš praktické. V porovnání s CSSM nabízí Variance Shadow Maps (VSM) podporu předfiltrování pomocí Čebyševovy nerovnosti při použití mnohem menšího množství paměti. Bohužel VSM samo nedovoluje korektně předfiltrovat průměrnou hloubku blockeru. Proto se využívá výběru vzorků hrubou silou. Křivka rekostrukce pro VSM je vidět na obr. 2.8a. Je snadné vidět, že tato funkce řeší jen jednu stranu funkce pro porovnávání stínu, a že je pro $(d - z) \leq 0$ je nedefinována. Většina existujících technik jednoduše přepokládá, že hodnota stínu je v této části rovna jedné. Pokud průměrná hodnota hloubky z_{avg} daného jádra je větší nebo rovna hodnotě hloubky d aktuálního bodu, tento bod bude určen jako plně osvětlený. Při práci s tvrdými stíny nebo pokud je jádro filtru velmi malé je tento předpoklad přijatelný. Pokud se však pracuje s velkým jádrem pro měkké stíny, pak předpoklad, že celé jádro bude osvětleno, může vést k nesprávným výsledkům (osvětlené pixely místo polostínu). Toto se v některých článcích označuje jako tzv. „non-planarity“ problém, do čestiny lze nejlépe přeložit jako nenáležitost do roviny, jelikož toto značí, že body jádra neleží ve stejné rovině. Takto špatně osvětlených artefaktů přibývá s rostoucí velikostí jádra.



(a) Funkce VSM

(b) Funkce CSM [3]

Obrázek 2.8: Porovnání mezi funkcemi pro předfiltrování stínů. d je hloubka stávajícího bodu a z je hloubka vzorku ze stínové mapy. [16]

Popis algoritmu

Přehled jednotlivých kroků algoritmu je vidět v algoritmu 2. Řádky z tohoto přehledu budou referovány jako (Řxx). Prvně se vygeneruje běžná stínová mapa dle VSM (Ř2), bude tedy obsahovat hloubku a její mocninu. Na základě této mapy se vygeneruje integrálový obraz (SAT) a min-max hierarchická stínová mapa (Ř3-Ř4). Poté se vykreslí scéna z pohledu kamery (Ř5) a pro každý viditelný bod P se nejdříve zjistí počáteční jádro filtru w_i (Ř7) protnutím roviny stínové mapy s jehlanem vytvořeným z bodu P a zdroje světla. Následně se zjistí průměrná hloubka z_{avg} v jádře w_i pomocí integrálního obrazu a min-max rozsah hloubky pomocí min-max HSM. Porovnáním hodnoty hloubky d bodu P s min-max rozsahem lze rychle určit plně osvícené a plně zastíněné body scény a lze je tedy ignorovat z dalších výpočtů měkkých stínů (Ř10).

Pro body scény, které ještě zbývají a které jsou tedy potenciálně v polostínu se ověří, jestli je jádro w_i „non-planarity“ nebo není. Podmínkou zde je $z_{avg} \geq d$. Pokud w_i není „non-planarity“ jádro, průměrná hloubka blockeru bude vyhodnocena přímo použitím rovnic (2.8). Pokud w_i je „non-planarity“ jádro, je potřeba toto jádro rovnoměrně nebo adaptivně rozdělit (Ř12) pro spočtení průměrné hloubky blockeru (Ř13). Více k principům dělení jádra viz 2.2.4.

Po získání průměrné hloubky blockeru lze zjistit vlastní jádro polostínu w_p (Ř16). Spočtení velikosti tohoto jádra v tomto kroku je podobné jako Ř7, jen rovina stínové mapy je nahrazena rovinou průměrné hloubky blockeru. Nakonec lze hodnotu variance shadow mapy pro jádro polostínu w_p spočíst buďto přímo nebo použitím dělení jádra.

Algoritmus 2 Přehled VSSM

```
1: Vykreslit scénu ze středu světla
2:    Vykreslit normální variance shadow mapu
3:    Vygenerovat integrální obraz (SAT)
4:    Vygenerovat min-max hierarchickou stínovou mapu (HSM)
5: Vykreslit scénu z pohledu kamery
6: Pro každý viditelný bod P:
7:    Spočíst počáteční šířku jádra w_i (blocker search area)
8:    Ověřit, jestli P není v plném stínu nebo plném světle pomocí HSM
9:    if(P je osvětleno nebo plný stín)
10:      Vrátit odpovídající hodnotu stínu
11:   if(w_i je „non-planarity“ jádro)
12:      Rozdělit jádro filtru
13:      Určit průměrnou hloubku blockeru použitím rovnice (2.8)
14:   else
15:      Určit průměrnou hloubku blockeru použitím rovnice (2.8)
16:   Spočíst jádro polostínu w_p založené na průměrné hloubce blockeru
17:   if(w_p je „non-planarity“ jádro)
18:      Rozdělit jádro filtru a určit hodnotu měkkého stínu
19:   else
20:      Určit hodnotu měkkého stínu přímo
21: Vykreslit finální obraz s použitím hodnot viditelnosti
```

Průměrná hloubka blockeru

Hlavním problémem při snaze napasovat VSM do kostry PCSS je efektivní určení průměrné hloubky blockeru, tedy první krok v PCSS.

Nechť w je jádro filtru a t je hloubka aktuálního bodu, předfiltrovanou hodnotu hloubky z a její mocninu z^2 lze získat z VSM. Dle lineárního filtrování je vzorek hloubky z již aktuální průměrnou hodnotou hloubky z_{avg} ve w. Hodnoty hloubky pro všechny texely ve w lze rozdělit na dvě samostatné kategorie: 1. hodnoty hloubky větší nebo rovny t, jejich průměr je pak definován jako z_{unocc}, 2. hodnoty hloubky, které jsou menší než t a jejich odpovídající průměr je definován jako z_{occ}. Za předpokladu, že jádro filtru w celkově obsahuje N vzorků, pak N_1 z nich je $\geq t$ a N_2 z nich je $< t$ a tedy platí rovnice (2.6).

$$\frac{N_1}{N} z_{unocc} + \frac{N_2}{N} z_{occ} = z_{avg} \tag{2.6}$$

Lze vidět, že $\frac{N_1}{N} z_{unocc}$ a $\frac{N_2}{N} z_{occ}$ odpovídají výsledkům stínového testu $P(x \geq t)$ a $P(x < t) = 1 - P(x \geq t)$. Tedy rovnici (2.6) lze přepsat následovně:

$$P(x \geq t) z_{unocc} + (1 - P(x \geq t)) z_{occ} = z_{avg} \tag{2.7}$$

Tedy průměrná hloubka blockeru z_{occ} se získá takto:

$$z_{occ} = \frac{z_{avg} - P(x \geq t) z_{unocc}}{(1 - P(x \geq t))} \tag{2.8}$$

Hodnota z_{avg} je známa a $P(x \geq t)$ lze určit na základě Čebyševovy nerovnosti. Jedinou neznámou zůstává průměrná hodnota hloubky pro nezastíněné části z_{unocc}. Pro případ s jedním roviným blckerem vrhající stín na rovnoběžnou rovinu, by $P(x \geq t)$ bylo přesné a tedy $z_{unocc} = t$. Z tohoto důvodu lze $z_{unocc} = t$ použít jako obecné řešení, bez významného zhoršení výsledné kvality stínu.

Zatímco rovnice může vypadat přímočaře, jejím základem je hodnota stínu z VSM, $P(x \geq t)$. Pokud je $z_{avg} \geq t$, pak se poruší podmínka Čebyševovy nerovnosti a opět se objeví „non-planarity“ problém. Pro správné určení průměrné hloubky blockeru, je nutné upravit $P(x \geq t)$, čehož lze docílit rozdělením jádra.

Řešení „non-planarity“ problému

Pro jakékoliv jádro filtru w bodu scény P vzniká „non-planarity“ problém pokud je $z_{avg} \geq t$, kde t je hloubka d aktuálního bodu.

Algoritmus VSSM navrhuje vyřešit „non-planarity“ problém pomocí dělení jádra w na množinu menších jader $\{w_{ci}, i \in [1...n]\}$. Podle toho, zda platí $z_{avg} \geq d$ pro w_{ci}, lze všechny podjádra rozdělit do dvou kategorií. Pro normální podjádra splňující podmínku $z_{avg} < t$, Čebyševova nerovnost sedí a lze určit průměrnou hloubku blockeru a hodnotu měkkého stínu na ní závislého. Pro podjádra s „non-planarity“ problémem, splňující podmínku $z_{avg} \geq t$, existují dvě možnosti. Lze předpokládat, že jsou všechny osvětleny, nebo lze použít krok PCF pro získání vzorků a určit stín. První možnost je obdobná předchozímu přístupu z VSM, jelikož je však podjádro w_{ci} mnohem menší než původní jádro w, „non-planarity“ problémem se efektivně potlačí. Druhá možnost je relativně laciná, 2x2 vzorků pro PCF není moc a mělo by zajistit dostatečnou přesnost.

Krajní body počátečního jádra w jsou známy, lze tedy celkem snadno rozdělit toto jádro na podjádra o stejné velikosti. Jak je vidět na obr. 2.9a, celý čtverec reprezentuje právě

počáteční jádro w a každý čtverec v mřížce reprezentuje podjádro w_{ci}. Smyčkou přes všechny podjádra se určí, která obsahují „non-planarity“ problém a která ne. Na obrázku 2.9a modrá barva znázorňuje „non-planarity“ problém a zelená normální jádra. Pro další použití se tyto jádra teoreticky rozdělí do dvou skupin, normální jádra w_{cj} a jádra s „non-planarity“ problémem w_{ck}.

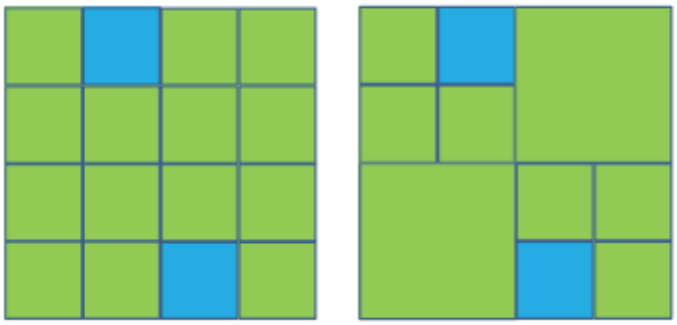
Nejdříve se uvažuje skupina w_{cj} normálních jader. Pro spočítání $P(x \geq t)$ pro celou skupinu, pomocí rovnice (2.5), je třeba určit střední hodnotu μ a varianci σ^2. Přesněji je třeba spočíst $E(x)$ a $E(x^2)$ ze všech podjader v této skupině. Definuje se velikost podjádra w_{cj} jako T_{cj} a získá se rovnice (2.9) pro střední hodnotu μ a varianci σ^2.

$$\begin{aligned} \mu &= \frac{\sum_j (E(x)_{cj} * T_{cj})}{\sum_j T_{cj}} \\ \sigma^2 &= \frac{\sum_j (E(x^2)_{cj} * T_{cj})}{\sum_j T_{cj}} - \mu^2 \end{aligned} \tag{2.9}$$

Během smyčky lze získat $E(x)_{cj}$ a $E(x^2)_{cj}$ z integrálního obrazu (SAT) pro každé jádro w_{cj}. Poté se spočte $E(x)_{cj} * T_{cj}$, suma všech hloubek texelů v jádře w_{cj}, stejně tak se spočte i $E(x^2)_{cj} * T_{cj}$. Nakonec se sumy hloubky a mocnin hloubky podělí sumou velikostí jádra (2.9). Tudíž lze vyčíslit rekonstrukční funkci VSM. Průměrnou hloubku blockeru d_1 ve skupině normálních podjader lze určit rovnicí (2.8).

Pro skupinu podjader s „non-planarity“ problémem lze použít klasický přístup PCF vzorkování pro každé w_{ck}. Vezme se m vzorků z w_{ck} a spočte se suma všech vzorků hloubky. Jelikož je jádro w_{ck} malé, $m = 2 * 2$ by mělo být dostačující. V dalších krocích je nashromážděna jak suma všech hloubek blockeru, tak i suma velikostí podjader. Podobně jako pro předchozí skupinu lze získat průměrnou hloubku blockeru d_2 pro skupinu s „non-planarity“ problémem.

Výslednou hloubku blockeru d pro celé jádro w lze spočíst jako vážený průměr d_1 a d_2, kde jako váha poslouží odpovídající velikosti podjader.



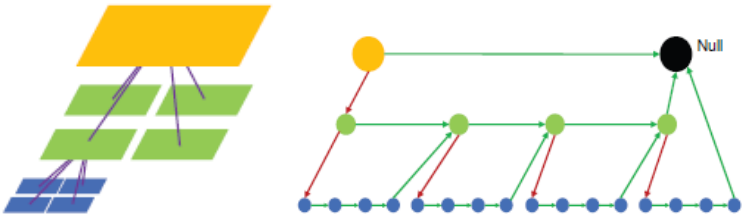
(a) Rovnoměrné dělení jádra (b) Adaptivní dělení jádra

Obrázek 2.9: Ukázka rovnoměrného a adaptivního dělení jádra o rozměrech 4×4. Zelená značí normální jádro, modrá je „non-planarity“. [16]

Pro zlepšení dělení jádra uvádí VSSM schéma pro adaptivní dělení (obr. 2.10). V porovnání s rovnoměrným dělením jádra, adaptivní dělení zpracovává podjádra hierarchicky. Výkon tohoto přístupu záleží na rovnováze mezi ziskem z hierarchického řezu a cenou za jeho průchod. Obecně pro velké množství podjader (≥ 64) je tento přístup výhodnější.

Jelikož jádro je vždy dvourozměrný čtverec, lze sestavit quadtree, viz obr. 2.10a. Pro jádro filtru w kořenový uzel stromu reprezentuje samotné jádro w a každý uzel stromu reprezentuje podjádro $\{w_{ci}, i \in [1...n]\}$. Nutné zdůraznit, že různé w_{ci} již nemají stejnou velikost a mohou tedy existovat na různých úrovních stromové hierarchie. V algoritmu 3 jsou ukázány kroky, vedoucí k vypočítání finální hodnoty měkkého stínu právě pomocí adaptivního dělení.

Klasický průchod stromem závisí na rekurzivních operacích, které není snadné naimplementovat pro GPU, které nemá zásobník. VSSM si půjčuje nápad řešení z článku [5] a vytváří tak lineární průchod stromem pro dvourozměrné jádro schopný fungovat na grafickém jádře. Pro dosažení lineárního průchodu každý uzel stromu musí mít definovány dva ukazatele, viz obr. 2.10b: ukazatel „Child“ (červená), ukazatel „Next“ (zelená). „Child“ ukazuje na první dceřiný uzel, „Next“ ukazuje na následují uzel v lineárním průchodu. Po pečlivém nastavení ukazatelů „Next“ pro každý uzel stromu [5], se lze vyhnout rekurzivnímu zpracování a lze provést lineární dopředný průchod pomocí GPU.



(a) Hierarchie jádra (b) Lineární průchod stromem

Obrázek 2.10: Lineární průchod stromem nad dvourozměrným filtrem. [16]

Výpočet hodnoty měkkého stínu potřebuje plně nasvícená podjádra. V algoritmu 3 je definována proměnná $LitArea$ zaznamenávající velikost všech plně osvícených podjader. Na řádcích (Ř9-Ř10), kde zaráz platí $E(x)_{ci} \geq d$ a $\sigma^2 < Threshold$, je stávající jádro w_{ci} plně osvětleno a všichni jeho potomci mohou být ignorováni. Hodnota $Threshold$ je konstantní pro celou scénu, dle VSSM by měla být rovna $0.00001 * r$, kde r je poloměr kulové obálky scény. Po získaní $LitArea$, lze přejít k dalšímu uzlu (Ř11). Pokud platí $E(x)_{ci} < d$, pak w_{ci} je normální podjádro. Jako předtím se nahromadí suma $E(x)_{ci}$ (Ř14), suma $E(x^2)_{ci}$ (Ř15) a suma T_{ci} (Ř16) a poté lze přejít k dalšímu uzlu (Ř17). Vyjímaje předchozí dva případy, posledním případem je situace, kdy platí $E(x)_{ci} \geq d$ a $\sigma^2 \geq Threshold$. V tomto případě je nutné vzít v potaz, zda je stávající uzel stromu i jeho listem nebo ne. Pokud stávající uzel není listem, pak se sestoupí ve stromu dolů na jeho potomka (Ř20). Jinak pokud je stávající uzel listem, lze použít PCF pro spočtení viditelnosti (Ř22-Ř25). Po výpočtech PCF lze přejít k dalšímu uzlu. Poté co se dokončí celý průchod stromem, je možné spočíst celkovou viditelnost (Ř31-Ř33). Nutné zmínit, že všechny tři vlastnosti podjader ($VSMArea$, $LitArea$ a $PCFArea$) jsou potřeba pro závěrečný krok.

Algoritmus 3 Adaptivní dělení jádra

Definuj $VSMEx = 0$, $VSMEx2 = 0$, $VSMArea = 0$
Definuj $PCFArea = 0$, $PSFBArea = 0$, $LitArea = 0$
Začít od kořenového uzlu jádra w, TreeNode = root
while TreeNode != Null **do**
    Stavající uzel je TreeNode w_{ci}
    Vypočítat texturové koordináty UV a velikost jádra T_{ci}
    Určit $E(x)_{ci}$ a $E(x^2)_{ci}$ ze SAT
    Vypočítat varanci σ^2
    if $E(x)_{ci} \geq d$ && $\sigma^2 < Threshold$ **then**
        $LitArea = LitArea + T_{ci}$
        TreeNode = TreeNode.Next
    else
        if $E(x)_{ci} < d$ **then**
            $VSMEx = VSMEx + E(x)_{ci} * T_{ci}$
            $VSMEx2 = VSMEx2 + E(x^2)_{ci} * T_{ci}$
            $VSMArea = VSMArea + T_{ci}$
            TreeNode = TreeNode.Next
        else
            if TreeNode není listem stromu **then**
                TreeNode = TreeNode.Child
            else
                Získat m vzorků uvnitř jádra w_{ci} v TreeNode
                $PCFArea = PCFArea + T_{ci}$
                $PCFBArea = PCFBArea + T_{ci} * \bar{m}/m$
                $//\bar{m}$ je počet zastíněných vzorků
                TreeNode = TreeNode.Next
            end if
        end if
    end if
end while
Vypočítat hodnotu rekonstrukce stínu L z $VSMEx$ a $VSMEx2$
Vypočítat viditelnost $L1$ dle $PCFArea$ a $PCFBArea$
Výsledná viditelnost se spočte pomocí L, 1.0 a $L1$, s váhami: $VSMArea$, $LitArea$ a $PCFArea$

Kapitola 3

Realizace algoritmu stínových těles

Tato kapitola se zaměří na samotnou implementaci algoritmu stínových těles viz oddíl 2.1. Popíší, na které části je třeba si při programování tohoto algoritmu dát pozor. Rozeberu případně další algoritmy nezbytné k naprogramování této techniky stínů.

Tento algoritmus je relativně přímočarý. Již dle prvního pohledu je jasné, že bude třeba provádět relativně velké množství výpočtů nad samotnou geometrií objektů scény. Lze tedy předpokládat, že hlavním stavebním kamenem při implementaci bude jednoduchý a rychlý přístup ke geometrii.

Z popisu algoritmu lze též vyvodit, že bude zapotřebí generovat velké množství částečných stínových map (rasterizace wedge), které budou skládat výslednou stínovou mapu.

3.1 Potřebné techniky

Sestavení celkového algoritmu měkkých stínu vyžaduje v základě dva další menší algoritmy. Prvním je získaní tvrdých stínu, které se získá pomocí stínových obdélníků (quadů). Další součástí je nalezení siluety objektu.

3.1.1 Tvrdé stíny

Sestavení tvrdých stínů je relativně postarší algoritmus. Už v roce 1977 jej například uvedl Franklin C. Crow [7]. V různých variantách se používá dodnes. V roce 1991 pak T. Heidmann ukázal, jak lze stíny realizovat pomocí stencil bufferu, čímž se dosáhlo použitelnosti v realtime aplikacích.

Sestavení tvrdých stínů pro polygonální model spočívá v projití všech polygonů a určení, které jsou natočeny na světlo, následně se všechny hrany protáhnou do nekonečna. Nekonečnem se prakticky bere taková vzdálenost, aby tyto obdélníky pokryly celou viditelnou část scény. Nesmí se též zapomenout na přední a zadní část, aby se dosáhlo uzavřené geometrie. Přední uzávěr může být tvořen samotným objektem, zadní lze občas vynechat v závislosti na použité variantě.

1: Najít hrany siluety
2: Protáhnout všechny hrany siluety ve směru zdroje světla
3: Přidat přední a zadní uzávěr

Existují tří základní varianty této techniky: depth pass, depth fail a xor. Všechny používají výše zmíněný postup pro sestavení stínového tělesa. Liší se však využití stencil bufferu.

Depth pass je založen na dvou samostatných průchodech, ve kterých se pomocí stencil bufferu počítá, kolik ploch je natočených dopředu, a kolik je odvrácených. Pokud by bod (plocha, či objekt) byl ve stínu, pak bude existovat více stínových ploch přivrácených k pozorovateli. Pokud je počet přivrácených a odvrácených shodný, řešený bod není ve stínu. Vytvoření stencil masky funguje tedy následovně:

1: Zakázat zápis hloubky a barvy
2: Zapnout ořezávání zadních stěn
3: Nastavit stencil operátor na přírůstek při depth pass
4: Renderovat stínové těleso
5: Přepnout na ořezávání předních stěn
6: Nastavit stencil operátor na dekrement při depth pass
7: Renderovat stínové těleso

Místa ve stencil bufferu obsahující hodnotu nula pak říkají, že počet přivrácených a odvrácených stěn stínových těles je shodný a tedy daná místa jsou mimo stín. Tento přístup má problém pokud je pozorovatel umístěn ve stínu, jelikož pak vidí prvně odvrácenou stranu stínového tělesa, což prakticky invertuje výsledné stíny. Tomuto se dá předejít přidáním přední uzávěry. Lze též připočíst jedna za každé stínové těleso, ve kterém je kamera umístěna, což může být zbytečně náročné. Dalším problémem, který však v dnešní době ztrácí na závažnosti, je nedostatečná bitová kapacita stencil bufferu.

Depth fail na rozdíl od předchozí varianty počítá, kolik ploch se nachází za řešeným bodem, přetočení hloubky. Tímto se vyřeší problém kamery umístěné ve stínovém tělesu, ale přidává podmínku, že musí být přítomen zadní uzávěr. Zadní uzávěra je nezbytná jinak by chyběly stíny v místech, kde se stínové těleso blíží k nekonečnu.

1: Zakázat zápis hloubky a barvy
2: Zapnout ořezávání předních stěn
3: Nastavit stencil operátor na přírůstek při depth fail
4: Renderovat stínové těleso
5: Přepnout na ořezávání zadních stěn
6: Nastavit stencil operátor na dekrement při depth fail
7: Renderovat stínové těleso

Xor variantu lze použít pro aproximaci předchozích dvou. Nefunguje úplně správně pro překrývající se stínová tělesa, ale potřebuje jen jeden renderovací průchod a 1-bitový stencil buffer.

1: Zakázat zápis hloubky a barvy
2: Nastavit stencil operátor na XOR při depth pass
3: Renderovat stínové těleso

3.1.2 Nalezení siluety objektu

Přestože jsem v teoretické části uvedl, že vyhledání siluety objektu by nemělo býti úzkým hrdlem tohoto algoritmu až na případy použití modelů s vysokou hustotou polygonů, tak jsem přesvědčen, že je vhodné se nad tímto problémem pozastavit.

Nalezení siluety objektu ve vztahu ke světlu znamená nalezení všech hran, které spojují dva polygony (trojúhelníky) tak, že jeden je natočený na světlo a druhý je natočený od

světla. Pro implementaci z tohoto tedy vyplývá, že bude třeba uchovat seznam hran a k ním náležící trojúhelníky.

Samotný seznam hran by nebyl příliš užitečný pro vykreslení objektu jako takového, takže bude třeba tyto seznamy vhodným způsobem provázat. Případně udržovat seznam hran samostatně spojený se zbytkem geometrie jen pomocí referencí.

K samotnému nalezení siluety lze například použít brute-force přístupu. Lze projít všechny hrany a zjistit, zda splňují podmínku siluety, tedy jestli jeden ze dvou polygonů hrany je natočený na světlo a druhý od něj. Zatímco je toto velice snadné na implementaci, není to rozhodně nejsofistikovanější přístup. Za prvé se pro každý polygon několikrát opakovaně spočítá, zda je natočen na světlo, pokud by se toto neuchovávalo pro další použití, čímž by se implementace zkomplikovala a lze pak rovnou využít trochu lepšího přístupu.

O něco lepším řešením, které vychází z předešlé úvahy o použití hrubé síly, je projít přímo polygony. Tedy lze projít všechny polygony modelu, spočíst pro ně natočení ke světlu a to uchovat v datovém objektu spravujícím daný polygon například jako proměnnou typu boolean. Za předpokladu, že máme sestavený seznam hran, pak stačí projít postupně jednotlivé hrany a porovnat tyto boolean hodnoty přilehlých polygonů.

Dalším způsobem, který vychází z předchozích dvou, a který prakticky odzkouším, je použití seznamu hran s tím, že se objekt hrany rozšíří o jednu celočíselnou hodnotu tzv. počítadlo (angl. *counter*). Tato celočíselná hodnota bude udávat, zda je hrana siluetou či nikoli. Pro získání siluety se nejdříve postupně projdou jednotlivé polygony a spočítá se, zda je natočen na světlo. Pokud je, tak se ignoruje a vezme se další, pokud není natočen ke světlu, tak se počítadla hran náležících tomuto polygonu zvětší o jedna. Výslednou siluetu objektu pak lze snadno získat ze seznamu všech hran tak, že porovnáme hodnotu počítadla hrany vůči jedné.

Za předpokladu, že známe rovinu $\vec{R}$ daných polygonů, je zjištění zda je polygon natočen na určitý bod $\vec{B}$ relativně snadné. Stačí ověřit zda platí $\vec{R} \cdot \vec{B} > 0$.

3.1.3 Ořezávání

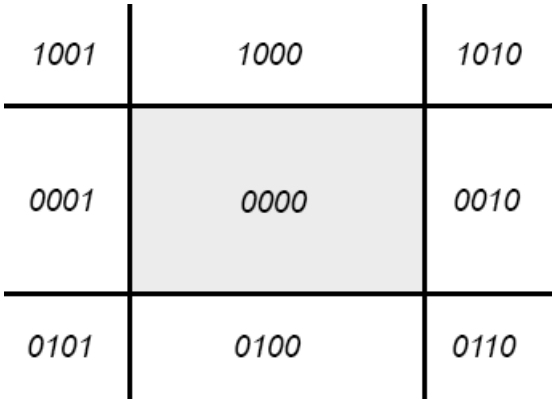
Techniky rozdělující obraz na vnější a vnitřní část se označují jako ořezávací. Oblast, vůči které je objekt ořezáván, je pak nazvána ořezové okno. V počítačové grafice jsou ořezové algoritmy hojně používané, přestože hlavně ořezávaní ve 3D není triviálním problémem. Díky těmto faktům existuje řada algoritmů, které se liši svým zaměřením, složitostí i využitím. Pro účely stínového algoritmu bude třeba ořezávat jistou úsečku pomocí obdélníkového okna. Pro tento účel se nabízí řada algoritmů. Pro srovnání rozdílu uvedu dva.

Algoritmus Cohen-Sutherland pracuje na principu oblastí. Koncovým bodům úsečky je přiřazen 4-bitový kód oblasti. Odpovídající bity jsou nastaveny v závislosti na pozici koncového bodu vůči ořezovému oknu (obr. 3.1). Pokud bod leží vlevo od okna je nastaven bit na první pozici, vpravo je nastaven druhý bit, pod oknem je nastaven třetí bit a pokud bod leží nad oknem nastaví se čtvrtý bit, názorně lze vidět na obr. 3.2. Při implementaci jsou toto čtyři podmínky.

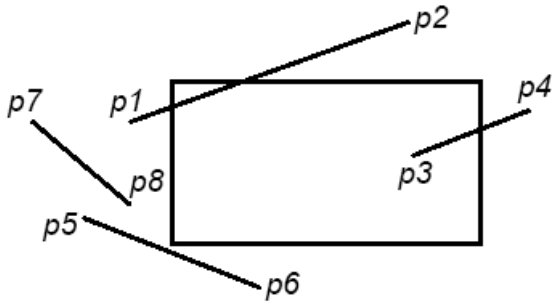
```
if(x < xmin)
  res |= LEFT;
if(x > xmax)
  res |= RIGHT;
if(y < ymin)
  res |= BOTTOM;
```

```
if(y > ymax)
  res |= TOP;
```

Prvním krokem je tedy zjistit kódy oblastí. Následně lze otestovat triviální přijmutí či odmítnutí. Lze udělat bitový AND, pokud je výsledek různý od nuly, tak úsečka leží celá mimo ořezové okno. Dále lze provést bitový OR, pokud je výsledek nulový, tak koncové body úsečky leží uvnitř okna. Pokud se nepodaří triválně zpracovat, tak se musí spočíst průsečíky s oknem. Pomocí bitových testů lze určit, kterou hranou okna se má ořezávat. Toto se opakuje dokud není úsečka celá v okně.



Obrázek 3.1: Bitové kódy jednotlivých oblastí pro ořezávání algoritmem Cohen-Sutherland.



Obrázek 3.2: Ukázka vyhodnocení úseček. p1: 0001, p2: 1000, p3: 0000, p4: 0010, p5: 0001, p6: 0100, p7: 0001, p8: 0001

Algoritmus Liang-Barsky využívá parametrických rovnic přímek a nerovností popisující rozsah ořezového okna k určení průsečíků přímky a ořezového okna. Pomocí těchto průsečíků se dá určit, které části přímky se mají vykreslit. Hlavní myšlenkou tohoto algoritmu je provést co nejvíce testování před samotným počítáním průsečíků. Jestliže parametrické vyjádření přímky vypadá následovně:

$$\begin{aligned} x &= x_0 + u(x_1 - x_0) = x_0 + u\Delta x \\ y &= y_0 + u(y_1 - y_0) = y_0 + u\Delta y \end{aligned} \tag{3.1}$$

Bod leží v ořezovém okně pokud platí:

$$\begin{aligned} x_{min} &\leq x_0 + u\Delta x \leq x_{max} \\ y_{min} &\leq y_0 + u\Delta y \leq y_{max} \end{aligned} \tag{3.2}$$

Toto lze vyjádřit jako čtyři nerovnosti:

$$up_k \leq q_k, k = 1, 2, 3, 4 \tag{3.3}$$

$$\begin{aligned} p_1 &= -\Delta x, \quad q_1 = x_0 - x_{min} \\ p_2 &= \Delta x, \quad q_2 = x_{max} - x_0 \\ p_3 &= -\Delta y, \quad q_3 = y_0 - y_{min} \\ p_4 &= \Delta y, \quad q_4 = y_{max} - y_0 \end{aligned} \tag{3.4}$$

Výslednou úsečku pak lze získat následovně:

1: Přímka rovnoběžná s hranou ořezového okna má $p_k = 0$ pro danou mez
2: Pokud pro toto k platí $q_k < 0$, tak je přímka kompletně mimo okno
3: Pokud je $p_k < 0$ přímka směřuje z vnějšku dovnitř okna a pokud je $p_k > 0$ přímka směřuje z okna ven
4: Pro nenulové p_k, $u = \frac{q_k}{p_k}$ udává bod protnutí
5: Pro každou přímku se spočte u_1 a u_2.
6: Pro u_1, kde $p_k < 0$, se vezme u_1 takové, aby bylo největší na intervalu $\left(0, \frac{q_k}{p_k}\right)$.
7: Pro u_2, kde $p_k > 0$, se vezme u_2 takové, aby bylo nejmenší na intervalu $\left(1, \frac{q_k}{p_k}\right)$.
8: Pokud je $u_1 > u_2$, přímka je mimo okno a nezpracovává se.

Liang-Barsky je pro vícerozměrné (3+) ořezávání mnohem efektnější než Cohen-Sutherland. Pro ořezávání ve 2D, které je relevantní pro algoritmus stínů, je Liang-Barsky rychlejší jen o pár procent. V shaderu se bude ořezávat jen jedna úsečka, bude tedy použit programově přehlednější Cohen-Sutherland.

3.2 Popis implementace

Prvním krokem, na který je třeba se zaměřit, je navrhnutí datové struktury uchovávající geometrii. Jak bylo zmíněno, nad geometrií objektu se budou provádět výpočty a není tedy vhodné, aby příliš bránila v přístupu k vnitřním hodnotám. Nebude též stačit uchovávat jen samotnou geometrii (vrcholy atd.), ale i jejich vzájemné vztahy (hrany apod.), případně některé informace navíc (je polygon natočený na světlo).

3.2.1 Základní datové objekty

Pro výpočty v algoritmu je třeba znát rovinu daného trojúhelníku (polygonu). Rovinu $\vec{R}$ lze získat snadno z normály $\vec{N}$ daného polygonu a jednoho bodu $\vec{B}$, náležícího tomuto polygonu, a to pomocí rovnice 3.5.

$$\vec{R} = (N_x, N_y, N_z, -\vec{N} \cdot \vec{B}) \tag{3.5}$$

Nyní tedy lze navrhnout základní podobu struktury pro základní jednotku geometrie (trojúhelník či polygon). Bude uchovávat jak jednotlivé vrcholy, tak rovinu, ve které leží, dále bude obsahovat seznam hran a identifikaci toho, zda je plocha aktuálně natočená ke světlu. Počáteční návrh v C++ stylu by vypadal takto:

```
class surface {
public:
  surface();
```

```
  vec3 vertices[N]; \\ N je konst pocet vertexu
  vec3 normal;
  vec4 plane;

  edge* edge[N];

  bool facingLight;
}
```

Takto navrhnutou třídu, lze rovnou rozšířit o jednu metodu. Přesněji o samotné vykreslení daného polygonu. Jelikož na této úrovni geometrie nejsou nikde uchovány transformace celého objektu (posun, atd.), je třeba si je předat, pokud s nimi bude chtít nějak pracovat.

```
void draw(vec3 *position);
```

V návrhu této třídy je uvedeno následující pole `edge* edge[N]`. Toto bude sloužit k uchování seznamu hran náležícího k danému polygonu. Obsah samotné třídy hrany se dá pak relativně snadno odvodit: dva body tvořící tuto hranu, ukazatele na dva polygony, které tato hrana spojuje a pro potřeby algoritmu hledání siluety počítadlo.

```
class edge
{
public:
  edge();

  vec3 e0;
  vec3 e1;
  surface *s1;
  surface *s2;
  int counter;
};
```

Tyto dvě třídy by tedy měly posloužit jako snadno přístupné datové struktury, nad kterými se bude dál pracovat s geometrií. K tomu bude zapotřebí třída uchovávající celý objekt. Tedy třída, která bude uchovávat pole či seznam jednotlivých polygonů, transformace objektu (např. posun) a další informace jako je materiál a seznam všech hran daného objektu, který pak poslouží pro vyhledání hran siluety. Je možné dopředu odhadnout několik metod, které bude objekt scény potřebovat. První je vykreslení daného objektu. Druhá metoda pak bude sloužit k zjistění hran siluety. Další metoda, která se dá odhadnout slouží k vykreslení geometrie generující tvrdý stín v prvním průchodu tohoto algoritmu. Výsledná třída objektu scény by tedy mohla vypadat následovně:

```
class sceneObject {
public:
  sceneObject(void);

  void draw(void);
  void drawShadowVolume(vec3 &light_pos);

  void calcSilhouetteEdges(vec3 &light_pos);
```

```
  vec3 position;
  surface *surfaces;
  material *m;
  list<edge*> edges;
};
```

3.2.2 Sestavení wedge

První část algoritmu v podstatě spočívá v získání stínové mapy obsahující tvrdé stíny. Jak získat tvrdé stíny jsem popsal již v předchozí části, implementaci není třeba do detailu probírat. Nutné je však podotknout, že výstupem tvrdých stínu má být taková maska, kde hodnota jedna je plné světlo a nula značí plný stín. Čehož lze dosáhnout tak, že pomocí stencil testu vykreslíme přes celý obraz plochu s požadovanou hodnotou, tedy nulou nebo jedničkou. Vykreslit plochu přes celý obraz je relativně snadné, stačí využít ortogonálního pohledu.

Masku z prvního průchodu bude třeba mít uloženou v textuře. OpenGL nabízí několik možností jak vyrenderovat scénu do textury. Jako první metodu lze scénu běžně vykreslit na stávající framebuffer (tedy snímek), následně vykopírovat framebuffer do textury, což není příliš rychlá operace dle manuálu OpenGL.

Lepším řešením je připravit si tzv. framebuffer object. Je to prostředek OpenGL, který umožnuje renderovat scénu bez nutnosti jejího zobrazení. Umožnuje určit, kam se bude scéna renderovat, lze použít i texturu jako výstup. Toto je v praxi rychlejší, jelikož se textura nemusí kopírovat zpět do operační paměti, ale pracuje se s ní přímo na grafické kartě.

Nyní se lze podívat na jednu ze dvou důležitých částí tohoto algoritmu. Sestavení wedge. Pro sestavení wedge je potřeba znát siluetu objektu. Silueta se dá zjistit již před tvorbou tvrdých stínů, což je i praktické, jelikož se jí dá využít právě pro sestavení tvrdých stínů.

Třída `surface` se rozšíří o metodu `drawWedges()`, která bude sestavovat všechny wedge pro daný polygon. Prvně se vezme původní hrana (`edge *e`), určí se vektory směru z obou krajních bodů hrany a spočítají se jejich délky.

```
vec3 vLightE0 = e->e0 - light_pos;
vec3 vLightE1 = e->e1 - light_pos;
float fLE0 = length(vLightE0);
float fLE1 = length(vLightE1);
```

Následně se určí, který bod je více vzdálen od světla a ten se posune na stejnou vzdálenost jako druhý bod. Zde bych se pozastavil nad jednou praktickou myšlenkou. Díky nepřesnosti při počítaní s floaty nebudou oba body nikdy v přesně stejné vzdálenosti, což není problém pro další výpočty.

Dle teorie k algoritmu lze pokračovat dál. Avšak ve skutečnosti tomu tak není. Musí se ošetřit připad, kdy jsou oba krajní body hrany vzdálené přesně stejně od světla. Jeden z bodů se musí mírně posunou ve směru ke světlu. Toto se musí udělat ze dvou důvodů. Prvním důvodem jsou výpočty, které budou následovat, a ve kterých by stejně vzdálené body dávali nesprávné výsledky. Druhým důvodem je to, aby původní hrana zaručeně prošla rasterizací. Pokud původní hrana bude náležet celá do wedge je toto zajištěno.

```
edge me(e->e0, e->e1);
if(fLE0 > fLE1){
```

```
  vLightE0 = normalize(vLightE0) * fLE1;
  me.e0 = light_pos + vLightE0;
}
else if(fLE1 > fLE0){
  vLightE1 = normalize(vLightE1) * fLE0;
  me.e1 = light_pos + vLightE1;
}
else{
  vLightE0 = normalize(vLightE0) * (fLE0 - 0.00001);
  me.e0 = light_pos + vLightE0;
}
```

Nově získané body hrany jsou výchozím bodem pro sestavení wedge. Jednotlivé roviny tvořící wedge by měly být co nejužší, aby se omezil počet nadbytečně zpracovávaných bodů při rasterizaci. Vezme se tedy v potaz geometrie světla. Pro světlo tvořené z quadu lze pak přední rovinu zjistit následovně. Postupně pro všechny body světla se pak zjistí normála trojúhelníku tvořeného modifikovanou hranou a jedním bodem světla.

```
for(int j = 0; j < 4; j++){
  vec3 front_normal = cross((me.e1 - lightVertices[j]),
                            (me.e0 - lightVertices[j]));
  front_normal = normalize(front_normal);
```

Za pomocí této normály se pak určí rovina, ve které se trojůhelník nachází.

```
  vec4 front_plane = vec4(front_normal,
                          dot(-front_normal, lightVertices[j]) );
```

U takto získané roviny je podstatné ověřit, zda je tato rovinou hledanou či nikoliv. Což lze provést tak, že se projdou ostatní body tvořící světlo a ověří se, že žadný z nich neleží za sestavenou rovinou.

```
  bool front_facing = true;
  for(int k~= 0; k~< 4; k++){
    if(k == j) continue;
    if(!(dot(front_plane, vec4(lightVertices[k], 1.0f)) > 0.0)){
      front_facing = false;
      break;
    }
  }
  if(front_facing) break;
}
```

Zadní rovinu lze získat téměř totožným způsobem. Jen se prohodí výpočet normály tak, aby směřovala na opačnou stranu nebo lze otočit podmínku v testování natočení roviny. Výsledek však nebude totožný. V prvním případě vyjde rovina, jež se dotýká světla v určitém bodě a ostatní body tvořící zdroj světla leží před touto rovinou. V druhém případě budou tyto body ležet za rovinou. Toto je důležité si uvědomit, jelikož se rovin bude využívat v shaderu pro testování náležitosti bodu do dané wedge.

Alternativně by bylo možné místo počítání rovin a ověřování, která rovina je správná, sestavit rovinu pomocí středu světla. Touto rovinou pak určit, které body leží za rovinou,

sestavit další rovinu pomocí jednoho z bodů a proces opakovat. Více méně by se jednalo o vyhledávání pomocí půlení intervalu, tento přistup mi podelší úvaze nepřišel příliš vhodný, jelikož by přinesl více problémů než užitku, kvůli nutným změnám v manipulaci s polem vrcholů světla.

Po získání přední a zadní roviny a jejich normál, lze přejít ke zjistění bočních rovin. Aby se dalo využít stejného principu jako pro předchozí dvě roviny, bude zapotřebí získat třetí orientační bod. Ten lze získat pomocí vektoru kolmého jak na modifikovanou hranu, tak i na vektor z krajního bodu hrany do světla. Bude tedy třeba určit tyto vektory.

```
vec3 vME0Light = light_pos - me.e0;
vec3 vME1Light = light_pos - me.e1;
vec3 vME0ME1 = me.e1 - me.e0;
vec3 vME1ME0 = me.e0 - me.e1;
```

Nyní lze sestavit pomocný bod pro pravou rovinu. Ta je definována tak, že obsahuje bod `me.e0` a dotýká se světla. Pro získání pomocného bodu se tedy využije dvou vektorů směřujících z tohoto bodu.

```
vec3 right_vector = cross(vME0Light, vME0ME1);
vec3 right_point = me.e0 + right_vector;
```

Nyní už lze pokračovat jako pro přední rovinu. Levá rovina se získá obdobným způsobem. Je definována přesně zrcadlově, pro sestavení pomocného bodu se tedy využije druhého bodu hrany.

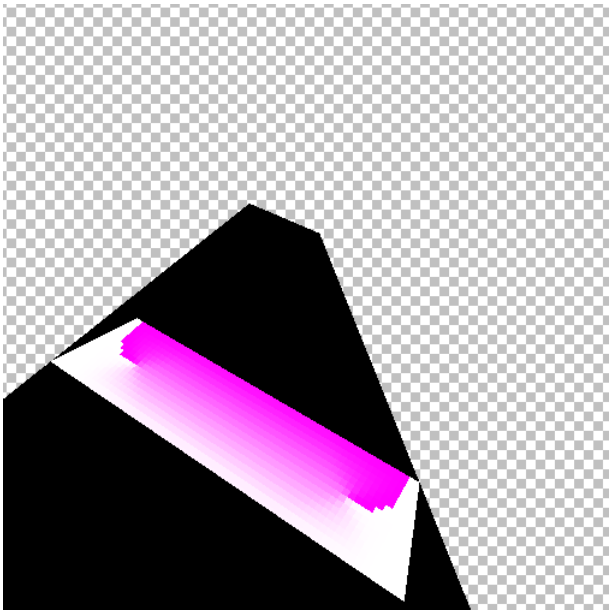
Získáním všech čtyř rovin lze považovat sestavení wedge za kompletní, tedy po matematické stránce. Pro další výpočty v shader programu toto stačí. Je však potřeba wedge sestavit pro vykreslení, aby bylo nad čím volat shader. Bude tedy zapotřebí z těchto rovin vytvořit plnohodnotné quady pro přední a zadní rovinu a trojúhelníky pro boční roviny. Je tedy třeba najít další čtyři body v „nekonečnu“.

Zde je na výběr ze dvou možných postupů. Lze postupně zjistit přímky tvořící průsečnice jednotlivých dvojic rovin. Tyto pak využít k nalezení potřebných bodů. Existuje však snažší a rychlejší řešení. Pro zjistění rovin bylo třeba zjistit prvně jejich normály. Po menším zamyšlení lze přijít na to, že vektor směru průsečnice dvou rovin je kolmý vůči oběma normálám daných rovin. Stačí tedy vektorově vynásobit jednotlivé normály. Je třeba si však dát pozor na pořadí násobení, jelikož pro určení bodu v nekonečnu jsou zajímavé jen vektory směřující od světla nikoli ke světlu. Případně podmínkou ověřit, zda je spočítán správný vektor, jestliže není, tak provést násobení v opačném pořadí. Ověřit toto lze například sestavením roviny pomocí spočtené normály a zkontrolováním, že střed světla leží za rovinou, nikoli před ní.

```
vec3 front_right = cross(front_normal, right_normal);
vec4 front_right_plane(front_right, dot(-front_right, me.e0));
if(dot(front_right_plane, vec4(light_center, 1.0)) > 0.0)
  front_right = cross(right_normal, front_normal);
front_right = normalize(front_right);
```

Pomocný bod v nekonečnu pak lze určit pomocí tohoto vektoru a příslušného krajního bodu hrany.

```
vec3 front_right_point = me.e0 + front_right*M_INFINITY;
```



Obrázek 3.3: Výsledek rasterizace jedné wedge.

Nyní lze tedy přejít k samotnému vykreslení wedge a její rasterizaci pomocí shaderu. V programu se tudíž zapne shader, který bude wedge rasterizovat, předají se mu jednotlivé roviny, modifikovaná hrana a případně další potřebné informace, které upřesním o něco později u samotného shader programu. Před samotným vykreslením wedge je ještě nutno se ujistit, že je vypnuté osvětlení a lze též zapnout ořezávání zadních stěn. Vykreslí se wedge, nejdříve přední a zadní stěna, následně pravá a levá stěna. Výsledek rasterizace jedné wedge lze vidět na obrázku 3.3.

Každá wedge se rasterizuje samostatně. Na vstup shaderu se posílá celá stínová mapa v aktuálním stavu a výsledkem je jen modifikovaná část. Pro získání celkové stínové mapy je tedy třeba jednotlivé rasterizované části spojit. Je tedy nutné překopírovat úpravu zpět do aktuální stínové mapy ještě před tím, než se bude rasterizovat další wedge. Toto lze provést jednoduchým shader programem.

3.2.3 Rasterizace wedge

Rasterizace se zrealizuje pomocí shader programu. Vertex shader obsahuje jen to nezbytné. Fragment shader obsáhne všechny výpočty. Jako první se ve fragment shaderu určí správná hloubka zpracovávaného pixelu pomocí textury s uloženou hloubkovou mapou scény. S takto získanou hloubkou a aktuální pozicí pixelu lze pomocí inverzní pohledové (modelviewprojection) matice získat zpětně pozici bodu ve „světě“.

```
vec2 coords = vec2(gl_FragCoord.x * (1.0/viewport.z),
                   gl_FragCoord.y * (1.0/viewport.w));
vec4 depth = texture2D(depthMap, coords);
vec4 pixel =
        vec4( ((gl_FragCoord.xy - viewport.xy)/ viewport.zw - 0.5) * 2.0,
              2 *(depth-0.5), 1.0);
pixel = iMVP * pixel;
pixel = pixel / pixel.w;
```

Při sestavení původní pozice pixelu je nutné upravit hloubku. Koordináty v rámci okna jsou na intervalu $< -1; 1 >$, včetně hloubky. Avšak textura s hloubkovou mapou uchovává hloubku na intervalu $< 0; 1 >$, proto je nutné ji převést zpět. Pokud by se na toto zapomnělo, stíny by byly nesprávně umístěny. Po získání pozice bodu ve světě lze přejít k samotnému počítání stínů. Nejdříve se zjistí, zda rasterizovaný bod leží uvnitř wedge nebo mimo, pak jej lze vynechat. Pro ověření, zda je bod uvnitř wedge se použijí roviny tvořící tuto wedge. Rasterizovaný bod musí ležet za rovinami, aby byl uvnitř wedge. U zadní roviny je třeba si dát pozor na to, jak byla sestavena a případně otočit test.

```
bool isInWedge(in vec4 p){
  if(dot(frontPlane, p) > 0.0)
    return false;
  if(dot(backPlane, p) > 0.0)
    return false;
  if(dot(rightPlane, p) > 0.0)
    return false;
  if(dot(leftPlane, p) > 0.0)
    return false;
  return true;
}
```

Pokud tedy bod leží uvnitř wedge je třeba jej zpracovat. Nejdříve se určí hodnota polostínu spočtením procentuálního zakrytí světla. Pro zjistění zákrytu světla je třeba provést projekci hrany do roviny světla. V případě, že hrany světla nejsou rovnoběžné s osami souřadného systému, pak je třeba světlo zpětně rotovat. Celou projekci je vhodné posunout tak, aby světelný zdroj ležel v kladné části a jeden z jeho bodů ležel v počátku souřadného systému. Tímto se zajistí, že se bude promítnutá hrana ořezávat pomocí os.

Promítnutí bodů hrany do roviny světla lze vyřešit několika způsoby. Jeden ze způsobů vychází z trigonometrie. Pozice rastrizovaného bodu a pozice bodu hrany jsou známé veličiny. Pro spočtení nové pozice body hrany je třeba spočítat o kolik se bod hrany posune od řešeného bodu. Spočítat vzdálenost rasterizovaného bodu od roviny světla je triviální. Rasterizovaný bod, průsečík roviny a kolmice vedené řešeným bodem udávají dva body pravoúhlého trojúhelníku. K získání třetího bodu je zapotřebí jedné informace navíc a to úhlu, který svírá přepona s odvěsnou u rasterizovaného bodu. Tento úhel lze získat právě pomocí bodu hrany. Spočíst délku přepony trojúhelníku a sestavení třetího bodu, hledaného promítnutí bodu hrany, je pak záležitost využití goniometrických funkcí. Funkce projekce by mohla vypadat přibližně takto:

```
vec3 project(in float b, in vec3 A, in vec3 dirB, in vec3 point){
  vec3 dir = normalize(point - A);
  float c = b / dot(dir, dirB);
  return A + dir*c;
}
```

Vstupní parametry jsou tedy vzdálenost od roviny (`b`), rasterizovaný bod `A`, směrnice kolmice (`dirB`) a bod, který se má promítnout (`point`). Promítnutí bodů hrany pro světlo ležící v rovině rovnoběžné s XZ by pak vypadalo následovně:

```
vec2 mE0 = project(d, point, vec3(0.0, 1.0, 0.0), e0);
vec2 mE1 = project(d, point, vec3(0.0, 1.0, 0.0), e1);
```

Za předpokladu, že hrany světla jsou již rovnoběžné s osami souřadného systému, posune se světlo do počátku snadno, odečtením jednoho z jeho bodů, takového který je nejblíže počátku.

Nyní lze pokračovat dvěma způsoby. Buďto se přesně spočítá plocha stínu, který hrana vrhá přes světlo. Lze využít matematicky elegantního řešení pomocí Greenova teorému nebo intuitivního přístupu, a pomocí stromu podmínek určit geometrický obrazec, který tvoří stín na světle a spočíst jeho plochu (prakticky tato plocha může mít tvar troj až šestiúhelníku). Obě metody jsou relativně náročné pro praktické použití v shader programu.

Druhý způsob je méně náročný na shader, ale za cenu možné ztráty přesnosti. Hodnoty se místo v shaderu předpočítají, buďto při startu aplikace (může být časově náročné) nebo samostatně mimo program a hodnoty se uchovají souboru textury. Ať tak či tak, do shader programu se předá textura obsahující množinu hodnot a na počtu těchto hodnot bude záviset kvalita výsledného stínu.

Právě pro tento přístup je třeba využít ořezávání zmíněné v předchozí části. Čtvercové (obdélníkové) světlo (či jeho obálka) srovnané s osami souřadného systému a umístěné v jeho počátku udává ořezové okno. K ořezání promítnuté hrany se použije os souřadného systému (tedy nul) a velikosti ořezového okna, což je vlastně nejvzdálenější bod od středu.

Samotná implementace ořezávání je víceméně přímočará. Určí se oblasti, ve kterých leží koncové body, pomocí čtyř jednoduchým testů.

```
int region(vec2 p, float xmin, float xmax, float ymin, float ymax){
  int res = 0;
  if(p.x < xmin) res |= LEFT;
  if(p.x > xmax) res |= RIGHT;
  if(p.y < ymin) res |= BOTTOM;
  if(p.y > ymax) res |= TOP;
  return res;
}
```

S takto získanými kódy oblastí se nejdříve provede bitový OR, pokud je výsledek různý od nuly (je nastaven některý bit) hrana leží celá mimo okno, zakrytí je tedy nulové. Následně se zkontroluje, zda hrana neleží celá v okně. Pak by se dala rovnou zpracovat.

```
if((first & second) != 0)
  return 0.0;
if(first == 0 && second == 0)
  return getCoverage(p1, p2);
```

Pro ostatní připady je nutné ořezávat. Prvně se zkontroluje, zda první koncový bod neleží uvnitř okna, hledal by se pak jen průsečík pomocí druhého bodu.

```
if(first == 0){
  inter = p1;
  firstdone = true;
}
```

Pokud je první koncový bod mimo okno je třeba najít průsečík s oknem. Průsečíky s levou a pravou stranou okna se spočtou stejně, jen s jinými mezemi (xmin→xmax).

```
if(!firstdone && (first & LEFT) != 0){
  inter.x = xmin;
```

```
  inter.y = ((xmin - p1.x)*(p2.y-p1.y)) / (p2.x - p1.x) + p1.y;
  if( inter.y >= ymin && inter.y <= ymax )
    firstdone = true;
}
```

Obdobně se určí průsečík s horní a spodní hranou okna. Opět budou oba výpočty podobné až na použití jiných mezí (ymax→ymin).

```
if(!firstdone && (first & TOP) != 0){
  inter.x = ((p2.x-p1.x)*(ymax-p1.y)) / (p2.y - p1.y) + p1.x;
  inter.y = ymax;
  if( inter.x >= xmin inter.x <= xmax )
    firstdone = true;
}
```

Toto vše se provede ještě jednou pro druhý koncový bod. Na konci pak vyjde oříznutá hrana.

Promítnutá a ořezaná hrana poslouží jako index do textury s předpočítanými daty. Hranu tvoří dva body ležící ve stejné rovině. Relevantní jsou tedy dva koordináty, celkově se tedy pro orientaci v textuře využijí čtyři souřadnice, což odpovídá čtyř rozměrné textuře (4D), kterou však prostředí OpenGL nepodporuje. Tento problém lze vyřešit namapovaním na běžnou 2D texturu, viz oddíl 3.2.4.

Nyní, po zjistění procentuálního zákrytu světla, se zjistí stávající hodnota stínové mapy jednoduše načtením z textury pomocí již spočtených koordinát (stejných jakých se využilo pro nalezení hloubky z hloubkové mapy). Dále je nutné určit, ve které části wedge bod leží. Wedge je rozdělena na dvě části. Kladnou a zápornou. Tyto části lze odlišit dvěma způsoby. Lze použít původní stínovou mapu s tvrdými stíny. Pokud je bod podle této mapy ve stínu, tak je v záporné části. Tento přístup by vyžadoval uchování textury a jedno čtení z této textury. Druhým způsobem je použití roviny quadu použitého pro vytvoření tvrdého stínu na stávající hraně. S pomocí této roviny lze snadno zjistit, jestli bod leží před rovinou, tedy v kladné části nebo za rovinou, tedy v záporné části.

```
bool isInPositiveHalfSpace(in vec4 p){
  return (dot(qPlane, p) > 0.0);
}
```

Pokud je bod v kladné části (původně plně osvětlena) je třeba zjištěná procenta zákryvu odečíst, pokud je v záporné části, tak přičíst.

```
if(isInPositiveHalfSpace(point))
  v = v - vp;
else
  v = v + vp;
```

V případě, že by se výsledné hodnoty z tohoto shaderu nevykreslovali do framebuffer objektu, bylo by třeba je upravit na interval $< 0; 1 >$, aby nedošlo k jejich nechtěnému oříznutí, které se provádí automaticky při přímém vykreslovaní na obrazovku. Při použití framebuffer objektu toto není třeba řešit, pokud je interní formát výstupní textury nastaven na float (plovoucí řádovou čárku), tak lze do ní zapsat jakoukoli hodnotu typu float. Shader se tedy dokončí zapsaním hodnoty. Jelikož je prakticky potřeba zapsat jednu hodnotu, lze

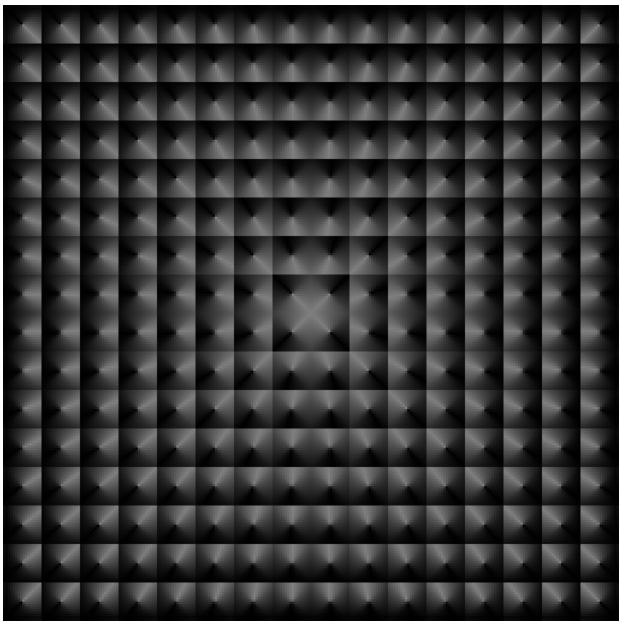
zbylé komponenty barvy použít k předání přídavných informací. Kupříkladu první hodnotou lze říci dalším shaderům ve zpracování (kopírovaní modifikované části do původní masky), že daný pixel byl upraven a je třeba jej zpracovat. Druhá komponenta barvy pak ponese samotnou hodnotu.

```
gl_FragData[0] = vec4(1.0, v, 0.0, 0.0);
```

Další komponenty by v případě nutnosti šlo využít k rozložení příliš velkých hodnot na vícero menších, ale v praxi by 32bitový float měl být více než dostatečný. V případě, že nebyl pixel vůbec zpracováván, jelikož leží mimo wedge, tak se zapíše na výstup sada nul.

3.2.4 4D textura

Prostředí knihovny OpenGL neumožňuje vytvoření a práci se čtyř rozměrnými texturami. Maximální velikost 2D textury je 4096x4096. Jestliže se má 4D textura namapovat na 2D, jinými slovy, jestliže se mají udělat řezy 4D textury a tyto řezy se mají uložit do roviny, dojde se k omezení rozměru 4D textury na $\sqrt{4096}$, tedy 64x64x64x64 vzorků.



Obrázek 3.4: Vyhledávací 4D tabulka rozložená do 2D textury. $n = 16$.

Dva koncové body promítnuté a ořezané hrany, x_1, y_1 a x_2, y_2 lze dohromady použít jako index pro čtyřrozměrnou vyhledávací tabulku. Tedy funkce $f(x_1, y_1, x_2, y_2)$ vrací zákryt světla podle dané hrany. Toto lze naimplementovat pomocí závislého čtení z textury, pokud se daná funkce f zdikretizuje.

Za předpokladu, že je zdroj světla diskretizován na $n \times n$ pozic texelů, a že první bod hrany se shoduje s některou s těchto pozic, např. $(x_1 = a, y_1 = b)$, kde a a b jsou celá čísla. Dalším krokem je vytvoření $n \times n$ podtextury, kde každá pozice texelu odpovídá koordinátům druhého koncového bodu hrany, x_2, y_2. V každém z těchto texelů se předpočítá hodnota vlastního zakrytí světla s použitím koncových bodů $(x_1 = a, y_1 = b)$ a x_2, y_2. Předpočítá se $n \times n$ oken s $n \times n$ podtexturami a uloží se do jediné dvourozměrné textury, která nahradí čtyřrozměrnou vyhledávací tabulku, ukázku pro $n = 16$ lze vidět na obrázku 3.4. Za chodu se pak spočte x_1, y_1 a zaokrouhlí se na nebližší střed texelu, který je použit pro

identifikaci, ve které $n \times n$ podtextuře se má vyhledávat. Druhý koncový bod je pak využit k přečtení hodnoty samotné.

Impelentace namapování 4D souřadnic na 2D souřadnice je sice teoreticky snadná, přece jen trochu zrádná. Texturové koordináty jsou v OpenGL typicky od nuly do jedné. Průsečíky jsou již z předchozích kroků upraveny na interval od nuly do velikosti světla. Nejdříve je tedy srovnáme na interval $< 0; 1$, prostým podělením velikostí světla.

```
point1 /= lightSize;
point2 /= lightSize;
```

Při mapování 4D textury na 2D, vznikne v podstatě matice řezů. Tedy při výběru bodu se musí dbát na přesnost. Musí se určit přesný počátek řezu, ze kterého se bude bod vybírat. Tento bod též musí být vybrán přesně. Nelze se spoléhat na filtrování přítomné v OpenGL, jelikož by výsledek byl zkreslený. Proto se souřadnice v intervalu $< 0; 1 >$ převedou na původní velikost textury v pixelech, čímž se výpočty přesunou do vyšších řádů a přebytečná desetinná místa lze oříznout. Následně se lze z celých čísel vrátit na desetinná místa, do intervalu $< 0; 1 >$. Pro bod řídící pohyb v řezu se tedy vydělí celkovou velikosti textury, bod řídící výběr řezu se vydělí počtem řezů vedle sebe.

```
point1.x = floor(point1.x * (dimension-1.0) + 0.5) / (dimension*dimension);
point1.y = floor(point1.y * (dimension-1.0) + 0.5) / (dimension*dimension);
point2.x = floor(point2.x * (dimension-1.0) + 0.5) / (dimension);
point2.y = floor(point2.y * (dimension-1.0) + 0.5) / (dimension);
```

Důležité je zmínit, že druhý bod je nutné též zdiskretizovat, jinak funkce vrací různé hodnoty zákrytu, když se prohodí koncové body. Při následném čtení lze též využít bilineárního filtrování.

Kapitola 4

Realizace algoritmu VSSM

Variance Soft Shadow Mapping je snahou o spojení různých metod pro zlepšení kvality měkkých stínů. Spojení těchto různorodých metod přináší kromě zlepšení stínů též vlastní problémy.

V této kapitole bych tedy rád uvedl informace potřebné pro implementaci VSSM algoritmu a popsal jednotlivé implementační části, ze kterých se tento algoritmus skládá.

4.1 Potřebné techniky

Měkké stíny realizované pomocí VSSM algoritmu využívají několika technik vypůjčených z různých algortimů pracujících se stínovými mapami. Za zmínění rozhodně stojí základní teorie k Summed Area Table. Hierarchické stínové mapy příliš teorie nemají, jedná se čistě o praktickou techniku, která je zmíněna či využita ve velkém množství různých algoritmů stínových map. O tom jak tedy vypadají se zmíním přímo u části popisující samotnou implementaci.

4.1.1 Summed Area Table (SAT)

Též známé jako *integrální obraz* (angl. *integral image*) je algoritmus pro rychlé a efektivní generování sum hodnot v obdélníkových podmnožinách dané mřížky. Poprvé byl uveden do počítačové grafiky v roce 1984 [8] pro použití v mipmapách, ale nebyl po mnoho let příliš používán. Avšak historicky jsou principy toho algoritmu dobře známy díky studijím v oboru funkcí pro vícerozměrné rozložení pravděpodobností.

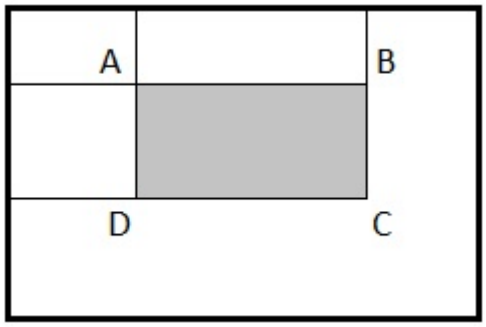
Jak naznačuje anglický název této techniky, hodnota jakéhokoliv pixelu (x, y) v integrálním obraze je prostě suma všech pixelů nad a vlevo od (x, y), včetně. Integrální obraz lze tedy zapsat rovnicí (4.1).

$$I(x, y) = \sum_{\substack{x' \leq x \\ y' \leq y}} i(x', y') \tag{4.1}$$

Integrální obraz lze získat jedním průchodem přes obraz pomocí rovnice (4.2) pro jeden bod obrazu.

$$I(x, y) = i(x, y) + I(x - 1, y) + I(x, y - 1) - I(x - 1, y - 1) \tag{4.2}$$

Jakmile je SAT jednou spočtena, získat obsah jakéhokoliv obdélníku lze v konstantním čase pomocí čtyř referenčních bodů. Například situaci na obrázku 4.1 popisuje rovnice (4.3).



Obrázek 4.1: Hledání sumy obdélníkové plochy

$$\sum_{\substack{A(x)<x'\leq C(x)\\ A(y)<y'\leq C(y)}} i(x',y') = I(C) + I(A) - I(B) - I(D) \tag{4.3}$$

4.2 Popis implementace

Techniky využívající stínové mapy nepotřebují znát nic o geometrii objektů scény. Oproti předešlému algoritmu stínových těles není tedy třeba vymýšlet jednoduché a rychlé přístupy ke geometrii objektů. Všechno zpracování se provádí pomocí shaderů. Jediné požadavky na aplikaci jsou tedy možnost vyrenderovat scénu z pohledu světla a aplikovat shader program na celou scénu.

Obecný postup pro renderování scény pomocí stínových map, lze aplikovat na algoritmus VSSM. Nejdříve se vyrenderuje scéna z pohledu ze světla, získá se hloubková mapa, ve VSM/VSSM případě hloubka a druhá mocnina hloubky (či rovnou momenty). Následně se vyrenderuje scéna z pohledu kamery, na kterou se aplikuje shader počítající stíny a osvětlení scény.

4.2.1 Implementace VSM

Nedílným základem algoritmu VSSM je Variance Shadow Map. Je tedy vhodné naimplementovat VSM už jen kvůli lepšímu pochopení VSSM. Navíc lze na tomto základu stavět dál při implementaci VSSM.

Při inicializaci aplikace se nachystá textura ve formátu float. Potřeba je i jeden framebuffer objekt, jehož výstupem bude právě přichystaná textura. Jako u všech algoritmů se stínovými mapami i zde se nejdříve vykreslí scéna z pohledu světla. Nastaví se tedy pohledové matice. Přepne se renderování do připraveného framebuffer objektu. Vykreslí se scéna s aplikovaným shaderem, který vytváří hloubku a mocninu hloubky, přesněji jeho výsledkem jsou přímo momenty potřebné pro další zpracování.

Shader počítající momenty je převážně implementován ve fragment shaderu, z vertex shaderu potřebuje jen předat na vstup polohu rasterizovaného bodu. Ve fragment shaderu se pak tato poloha použije ke zjištění hloubky pixelu.

```
varying vec4 v_position;
void main(){
  float depth = v_position.z / v_position.w;
```

Hloubka se nejdříve upraví z intervalu $< -1; 1 >$ na interval $< 0; 1 >$.

```
depth = depth * 0.5 + 0.5;
```

První moment tvoří hloubka jako taková, lze odvodit z rovnice (2.2).

```
float moment1 = depth;
```

Druhý moment z rovnice (2.3) tak snadný není. Nejdříve se určí druhá mocnina hloubky.

```
float moment2 = depth * depth;
```

Následně se určí parciální derivace hloubky, tyto se umocní a sečtou. Jedna čtvrtina tohoto součtu se pak přičte k mocnině hloubky a získá se tak druhý moment.

```
float dx = dFdx(depth);
float dy = dFdy(depth);
moment2 += 0.25*(dx*dx+dy*dy);
```

Nakonec se vypočítané momenty zapíší na výstup.

Dokud jsou dostupné pohledové matice pro render ze světla, je možné upravit matici jedné z aktivních texturovacích jednotek tak, aby reflektovala projekci ze světla.

```
glMatrixMode(GL_TEXTURE);
glActiveTexture(GL_TEXTURE7);
const GLdouble bias[16] = {
  0.5, 0.0, 0.0, 0.0,
  0.0, 0.5, 0.0, 0.0,
  0.0, 0.0, 0.5, 0.0,
  0.5, 0.5, 0.5, 1.0};
glLoadIdentity();
glLoadMatrixd(bias);
glMultMatrixd(lightProjectionMatrix);
glMultMatrixd(lightViewMatrix);
```

Pokud se uchovají pohledové matice světla pro pozdější použití není potřeba nastavovat texturovací jednotku a lze těmito maticemi upravit až koordináty v shaderu programu při čtení z textury.

Druhou částí implementace VSM je vykreslení scény z pohledu kamery. U klasického VSM by před vykreslením scény proběhlo například rozmazání pro získání měkčích stínů. Vykreslí se tedy scéna s aplikovaným shaderem, který je vlastně implementací Chebyševovy nerovnosti (2.5). Na vstupu z vertex shaderu se předají projekční koordináty, které se určí buď pomocí upravené texturovací matice nebo pomocí pohledových matic světla.

```
shadowCoordW = shadowCoord / shadowCoord.w;
float shadow = chebyshevUpperBound(shadowCoordW.z);
```

Takto získaným stínem lze modifikovat pixel scény.

Implementace funkce realizující Čebyševovu nerovnost jako takovou pak nejdříve zjistí načtením z textury pomocí projekčních koordinát momenty.

```
float chebyshevUpperBound(float dist) {
  vec2 moments = texture2D(shadowTex, shadowCoordW.xy).rg;
```

Pokud je hloubka bodu menší než první moment, tedy hloubka blockeru, je bod v plném světle.

```
if (dist <= moments.x)
  return 1.0 ;
```

Pokud tomu tak není, je fragment ve stínu či polostínu a je nutné spočíst varianci a následně určit p_{max} (2.5). Tedy jako moc je pravděpodobné, že daný pixel je osvícen.

```
  float variance = moments.y - (moments.x*moments.x);
  variance = max(variance,0.00002);
  float d = dist - moments.x;
  float p_max = variance / (variance + d*d);
```

Důležitý je řádek s `max(variance,0.00002)`, toto zajištuje, že se nebude počítat s příliš malými hodnotami variance a předejde se tak chybám vzniklých nepřesností operací s typem float na GPU.

4.2.2 Implementace SAT

Převést integrální obraz do podoby, která je snadno a rychle implementovatelná není triviální. Matematické podklady k mnou zvolené metodě implementace lze najít v článku [13].

Nejdříve se připraví dvě textury pro uchovaní mezikroků. Jelikož OpenGL nedovoluje zapisovat do stejné textury, ze které se čte, je toto nutné obejít pomocí dvou textur vázaných jako výstupy dvou různých framebuffer objektů. Lze též použít jeden framebuffer objekt a prohazovat texturu na jeho výstupu, avšak dle manuálu OpenGL je tato operace relativně pomalá.

Vstupem tohoto algoritmu je variance shadow mapa. Prvně je nutné se přepnout do ortogonálního zobrazení. Dále se spočítá kolik vertikálních a horizontálních průchodů je třeba provést, pomocí velikosti stínové mapy.

```
int nm = ceil(log2((float)SIZE));
```

Následují dvě totožné smyčky. První pro horizontální průchod a druhá pro vertikální. Na začátku opakování smyčky se změní framebuffer objekt.

```
for(int i=0; i < nm; i++){
  if (usingA) {
    glBindFramebuffer(GL_FRAMEBUFFER, fboA);
  } else {
    glBindFramebuffer(GL_FRAMEBUFFER, fboB);
  }
```

Právě použitému shaderu se předá aktuální krok pro sumaci a aktuální vstupní textura. Sumační krok N_i začíná na hodnotě jedna (2^i).

```
  shader->sendUniform1("Ni", (int)ni);
  shader->sendSampler2D("tex", atex);
```

Vymaže se stávající obsah framebuffer objektu a vykreslí se do něj nový mezikrok sumy, pomocí quadu přes celý obraz. Dále se posune sumační krok.

```
ni = ni << 1;
```

Prohodí se vstupní textury pro shader a přenastaví se proměnná určující, který framebuffer objekt je aktuální.

```
if (usingA) atex = 1;
else atex = 2;
usingA = !usingA;
```

Po skončení první smyčky pro horizontální průchod, se změní sumační krok zpět na hodnotu jedna, změní se aktivní shader program na vertikální průchod a provede se stejná smyčka znova.

Shader programy pro horizontální a vertikální průchod jsou téměř totožné. Jedná se o programy pro fragment shader. Na vstupu je šířka textury (či přímo velikost texelu) a sumační krok N_i. Nejdříve se určí aktuální texturové koordináty a z nich druhá sada koordinát posunutá o sumační krok.

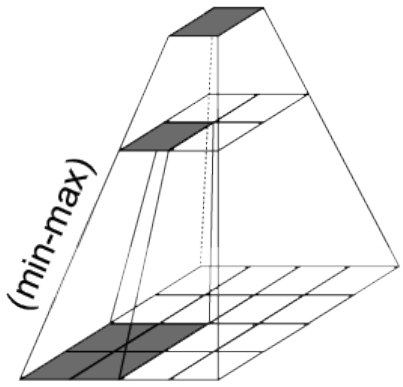
```
vec2 coords = gl_TexCoord[0].st;
vec2 coords2 = coords;
coords2.x = coords2.x + (1.0 / float(texWidth) * float(Ni));
```

Koordináty se posunují po x-ové ose pro horizontální průchod a po y-ové ose pro vertikální průchod. Takto získané souřadnice se použijí k přečtení dvou hodnot ze vstupní textury. Jejich součet se pak zapíše na výstup.

```
vec4 res = texture2D(tex, coords) + texture2D(tex, coords2);
gl_FragColor = vec4(res.x, res.y, res.z, 1.0);
```

4.2.3 Implementace HSM

Hierarchická stínová mapa je technika sloužící k zrychlení vyhledávání minimální a maximální hloubky scény. Původní stínová mapa se postupně zmenšuje vždy na poloviční velikost. Body menší verze stínové mapy vždy obsahují minimální a maximální hodnotu čtyř bodů z předchozí velikosti mapy. Princip je ilustrován na obrázku 4.2.



Obrázek 4.2: Min-max hierarchická stínová mapa

Prvním krokem při vytváření HSM je příprava mipmap textury, která bude sloužit pro uchování HSM. Místo mipmap lze využít N-buffer techniky [9]. Mipmapy lze však generovat velmi efektivně a potřebují málo texturovací paměti. Avšak chyby jimi vytvořené mohou

být viditelné, hlavně při čtení z větších úrovní. Oproti tomu N-buffer přístup nabízí přesnost za cenu delšího generování a využití více paměti.

Vytvoření mipmap textury v OpenGL není příliš odlišné od běžné textury, pro HSM je však třeba specificky nastavit některé vlastnosti. Jednou z prvních vlastností, které jsou nutné je chování textury při čtení mimo její hranice. Je nutné omezit přístup tak, aby se koordináty ořízly zpět na hranu texury.

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Též je důležité nastavit filtrování textury na nejbližší i v rámci mipmapy. Pro mipmap filtr je to nezbytné, u ostatních lze experimentovat, ale výběr nejbližšího prvku je jistotou.

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_NEAREST);
```

Typ textury, která bude sloužit pro uchování HSM, je víceméně zřejmý, bude se jednat o RGBA texturu ve float formátu.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
             SIZE, SIZE, 0, GL_RGBA, GL_FLOAT, NULL);
```

Pro dokončení mipmapy je nutné OpenGL říci, že má vygenerovat mipmapu pro tuto texturu. Toto lze učinit dvěma způsoby. První je validní i pro starší verze OpenGL, druhý je však preferovaný, jelikož odpovídá novým standardům.

```
//glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
glGenerateMipmap(GL_TEXTURE_2D);
```

Jelikož jednotlivé úrovně mipmapy budou naplněny vlastními daty, nevyužije se tedy možnosti automatického generování z OpenGL, je třeba zajistit rychlý zápis. Lze si tedy nachystat framebuffer objekt, jehož výstupem bude aktuální úroveň mipmap, do které se má zapisovat. Přehazování textury na výstupu není však příliš rychlé dle manuálu OpenGL. Některé webové zdroje uvádí, že přehození framebuffer objektu jako takového je až 20x rychlejší než prohození textury na výstupu framebuffer objektu. Proto je vhodnější připravit si sadu framebuffer objektů pro jednotlivé úrovně mipmapy. Zjistí se tedy počet úrovní mipmapy.

```
int nMips = ceil(log2((float)SIZE));
```

Následně se vygeneruje potřebné množství framebuffer objektů. Nesmí se zapomenout na nultou úroveň a v cyklu se nastaví výstup těchto framebuffer objektů na jednotlivé úrovně HSM mipmapy.

```
glGenFramebuffers(nMips+1, fboHSM);
for(int i = 0; i <= nMips; i++){
  unsigned int size = SIZE >> i;
  glBindFramebuffer(GL_FRAMEBUFFER, fboHSM[i]);
  glFramebufferTexture2D(GL_FRAMEBUFFER,
                         GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, hsm, i);
  glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Tímto je vše připraveno pro rychlý zápis do jednotlivých úrovní HSM. Aby se dalo zapisovat do HSM, je vhodné se přepnout do ortogonálního pohledu a vypnout osvětlení. Vstupem pro HSM je stínová mapa. Přesněji variance shadow mapa, která obsahuje jak hloubku tak i mocninu hloubky. Nultá úroveň mipmapy tvoří tedy kopii této variance shadow mapy. Vytváří se min-max HSM, cílem tedy je, aby v nejvyšší úrovni mipmapy byla nejmenší a největší hloubka scény.

Aktivuje se nultý framebuffer objekt a vymaže se jeho obsah, pokud se framebuffer objekt alespoň jednou nesmaže, jakýkoli zápis do něj se neprovede.

```
glBindFramebuffer(GL_FRAMEBUFFER, fboHSM[0]);
glClear(GL_COLOR_BUFFER_BIT);
```

Vykreslí se quad přes celý viewport s VSM jako texturou a vytvoří se tak kopie VSM. Další krok bude vytvořit vyšší úrovně mipmapy, o to se postará smyčka. Ještě před smyčkou se nastaví HSM textura jako aktivní texturovací jednotku, jelikož se z ní či z některé její úrovně bude číst pro získaní vyšší úrovně.

```
glBindTexture(GL_TEXTURE_2D, hsm);
for(int level = 1; level <= nMips; level++)
```

Obsahem smyčky je pak aktivace správného framebuffer objektu a nastavení parametrů pro mipmap část HSM textury tak, aby se dalo číst jen z předchozí úrovně.

```
glBindFramebuffer(GL_FRAMEBUFFER, fboHSM[level]);
glClear(GL_COLOR_BUFFER_BIT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, level-1);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, level-1);
```

Následně se spočte velikost texelu předešlé úrovně a pošle se do shaderu obstarávajícího sestavení jednotlivých úrovní mipmapy, o něm o něco později. Vykreslí se quad přes celý obraz s aplikovaným shaderem. Po provedení této smyčky lze vrátit nastavení mipmapy tak, aby se dalo číst ze všech jejích úrovní.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, nMips);
```

Po vrácení pohledových matic a zpětného zapnutí světla je generovaní min-max HSM dokončeno.

Shader vytvářející jednotlivé úrovně mipmapy není nikterak složitý. Veškeré operace jsou prováděny ve fragment shaderu. Zjistí se stávající texturové koordináty a načte se hodnota z předchozího stupně HSM.

```
vec2 coord = gl_TexCoord[0].st;
vec4 depth = texture2D(hsm, coord), depth2;
```

Postupně se načtou tři další body z nižší úrovně HSM a zjistí se z nich minimální a maximální hodnota.

```
coord.x += texel;
depth2 = texture2D(hsm, coord);
depth.x = min(depth.x, depth2.x);
depth.y = max(depth.y, depth2.y);
```

```
coord.y += texel;
...
coord.x -= texel;
...
```

Porovnání lze zduplikovat i pro `.z` a `.w`, čímž se zajistí minimální a maximální hodnoty pro mocninu hloubky. Tedy za předpokladu, že první dvě komponenty barvy v původní VSM uchovávají hloubku a druhé dvě komponenty mocninu hloubky, což se dá zajistit v shaderu pro VSM nebo lze použít triviálního shaderu při kopírování VSM do nulté úrovně HSM. Nakonec se tedy zapíšou nová minima a maxima do stávající úrovně mip-mapy.

Kapitola 5

Zhodnocení výsledků

V této kapitole uvedu zhodnocení algoritmů po stránce náročnosti na jejich nastudování a náročnosti na praktickou aplikaci takto získaných znalostí. Dále uvedu jednotlivé výhody a nevýhody těchto algoritmů. Popíši problémy, které tyto algoritmy obsahují a problémy, které vznikli při jejich implementaci.

5.1 Stínová tělesa

Algoritmy pro stínová tělesa nejsou v počítačové grafice žádnou novinkou. Mnou zvolená verze existuje již několik let. Pochopení jejího principu nevyžaduje příliš dodatečných znalostí, základní povědomí o principu stínových těles je více než dostatečné.

Problémem při studování tohoto algoritmu může být nemožnost získat některé ze starších článků o tvrdých stínech, na které se autoři referují a je tedy nutné použít jiný, ale přitom zachovat požadovaný výstup pro použití v prvním průchodu tohoto algoritmu.

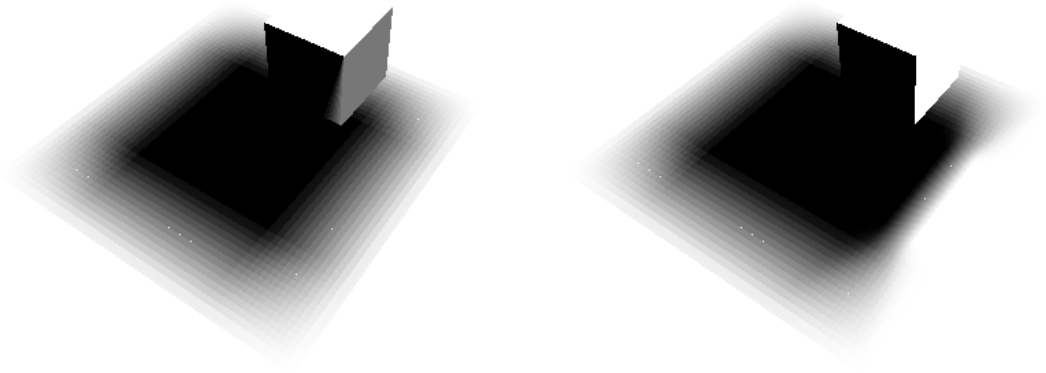
Dalším prvkem algoritmu, který v článku uvádějícím tento algoritmus není uveden, je matematické sestavení wedge. Jelikož je toto hlavní částí algoritmu, většinu výpočtů bylo třeba domyslet a odzkoušet, jestli dávají očekávané výsledky.

5.1.1 Aproximace siluety objektu

Zvolený algoritmus navrhoval použití aproximace siluety. Silueta objektu vrhajícího stín se vyhledává jen pomocí středu světla. Chybu vzniklou touto aproximací lze vidět na obrázku 5.1b.

Při použití jen jednoho bodu světla (středu) pro zjištění siluety je stín vypočítán správně, dokud je střed světla umístěn nad objektem. V momentě, kdy se střed světla přesune mimo objekt, se změní silueta. Avšak nově zjistěná silueta není korektní. Na obrázcích 5.1a a 5.1b lze pozorovat vrhaný stín při umístění středu světla blízko okraje objektu a těsně mimo objekt. Na 5.1b, kdy je střed světla mimo objekt, se sestaví wedge ze spodní hrany. Správně by se měla sestavovat z horní hrany, až do momentu, kdy celá plocha zdroje světla není mimo objekt.

Aproximace siluety pomocí jednoho bodu je možná rychlejší, ale cenou za rychlost je kvalita výsledného stínu. Zajímavé je, že i přes chybně sestavenou wedge, je vzniklý stín souvislý díky principu jakým se wedge rasterizuje.



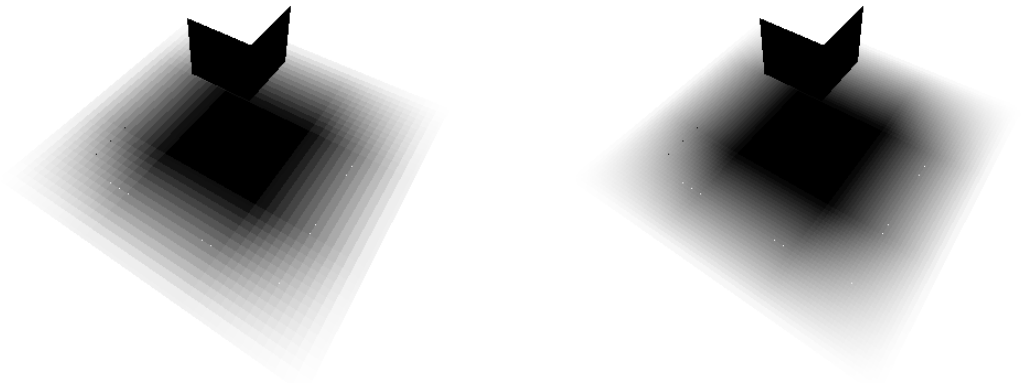
(a) Střed světla umístěn nad objektem. (b) Střed světla umístěn těsně vedle objektu.

Obrázek 5.1: Ukázka stínové mapy s chybou vznikající aproximací siluety.

5.1.2 Předpočítané hodnoty a kvalita stínu

Kvalita výsledného stínu je přímo úměrná velikosti 4D vyhledávací tabulky. Na obrázku 5.2 je vidět srovnání stínů při použití různých velikostí vyhledávací tabulky.

Využití předpočítané tabulky hodnot má jisté výhody. Dle vzdálenosti světla od objektů a podle vzdálenosti kamery může stačit menší vyhledávací tabulka. Čím dál je světlo od objektu, tím užší budou wedge polostínu a na jejich rasterizaci bude stačit méně hodnot z vyhledávací tabulky. Obdobně čím blíž se posune kamera k polostínu, tím více hodnot ve vyhledávací tabulce bude potřeba pro souvislý vzhled stínu. Pokud tedy není třeba šetřit místem v paměti, je vhodné u tohoto algoritmu použít co největší vyhledávací tabulku, ale už tabulka o $n = 32$ dává kvalitní stín, viz obrázek 5.2b.



(a) Vyhledávací tabulka $n = 16$. (b) Vyhledávací tabulka $n = 32$.

Obrázek 5.2: Rozdílné stíny pro různé velikosti vyhledávací tabulky.

5.1.3 Nepřesnost při výpočtech ve floatu

Grafické karty jsou přizpůsobeny práci s plovoucí desetinnou čárkou. Během implementace shaderu pro rasterizaci wedge jsem narazil na jeden zajímavý problém, týkající se nepřesnosti výpočtů v datovém typu float. Větší část shader programu jsem naprogramoval nejdříve pro CPU v jazyce C++ a po otestování jsem tyto části přepsal do jazyka GLSL pro shader grafické karty.

Hlavně u algoritmu pro ořezávání úseček, který ve své C++ implementaci naprosto bez problému fungoval a vracel správné výsledky, bylo nemilým překvapením, že jeho přímý přepis do GLSL nedělal vůbec nic. Při práci s dosti malými hodnotami totiž v GLSL dochází k značným nepřesnostem a i když se spočítá to samé několikrát, tak není jisté, že vyjde to samé.

Další nevýhodou těchto nepřesností je, že může vyjít nula tam, kde by správně vyjít nemohla a tím může nastat například dělení nulou, které v GLSL nemá definován výsledek, narozdíl od C++. Naopak v experimentu jsem zjistil, že tam kde mělo vyjít celé číslo, třeba i nula, skutečná hodnota byla někde v okolí tohoto čísla. Do shader programu se tedy hlavně v algoritmu pro ořezávání muselo dopsat několik úprav zajišťujících ochranu proti těmto nepřesnostem. Experimentálně jsem též zjistil, že přesnost výsledků je na jednu stotisícinu. V některých případech dokonce jen na desetitisícinu.

5.1.4 Chybně určený pozitivní/negativní prostor

Algoritmus rozděluje wedge na kladný a záporný poloprostor, ve kterých se aktuální hodnota stínu buďto odečítá nebo přičítá k celkové. Tento prostor lze přesně určit pomocí roviny quadu tvořícího tvrdý stín dané hrany. Nevýhodu tohoto přístupu lze pozorovat na obrázku 5.3.

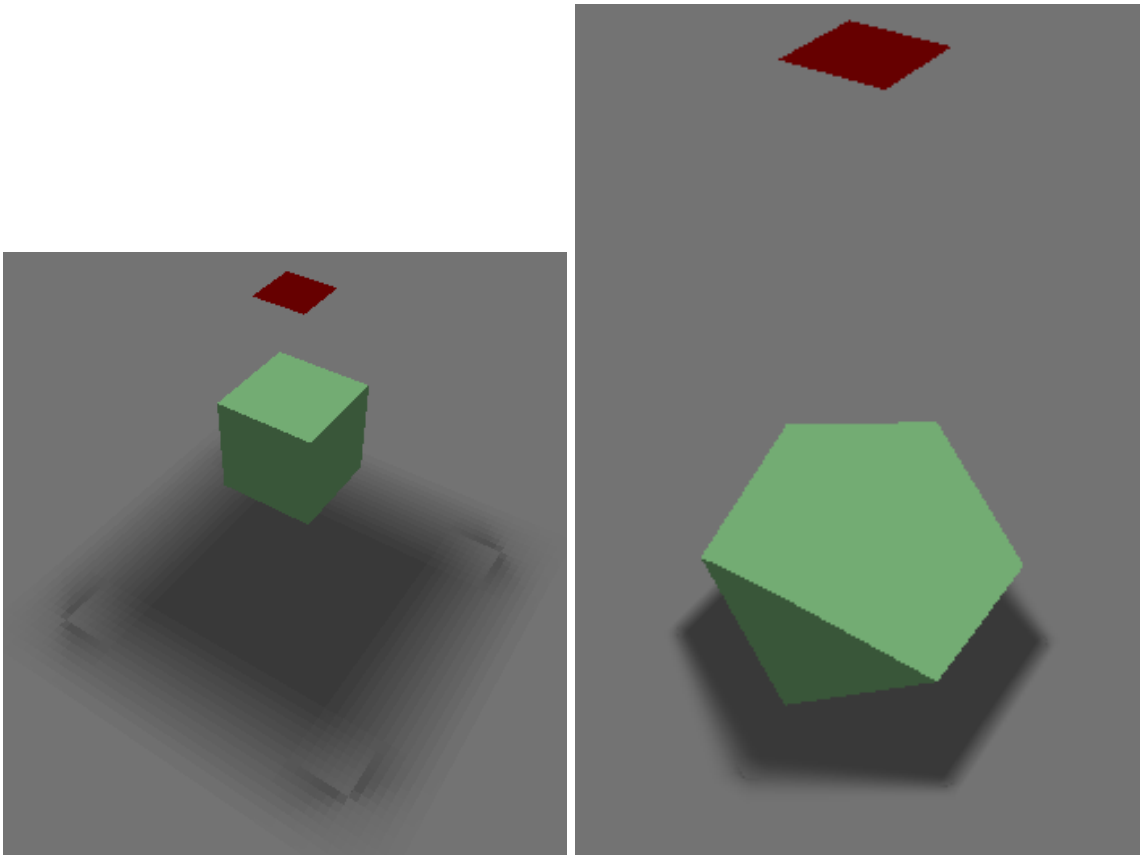
Tyto chybějící pixely vznikají díky již zmíněným nepřesnostem při práci s čísly v plovoucí desetinné čárce. V shaderu vyjde umístění před/za rovinou quadu jinak než při rasterizaci tohoto quadu v prvním průchodu.

Obrázek 5.3: Špatně určené pixely na rozmezí kladného a záporného prostoru wedge.

Problém lze vyřešit uchováním kopie původní stínové mapy z prvního průchodu a jejím použitím k určení v jaké části wedge se bod nachází. Využití původní stínové mapy však sebou přináší novou chybu, viz obrázek 5.4a. Tyto dva rozdílné přístupy totiž různě určí kladný a záporný prostor, rozdíl lze vidět na obrázku 5.5. Tato chyba však pro malé stíny není na první pohled tak patrná (5.4b), jako mizející pixely.

5.1.5 Nevýhody algoritmu

Hlavní nevýhodou tohoto algoritmu je jeho naprostá nefunkčnost pro více objektů, jejichž stíny se vzájemně překrývají. Autoři tohoto algoritmu tuto drobnou skutečnost zmínili v poznámce na konci své práce. Řešení tohoto problému není zrovna triviální. Pro další



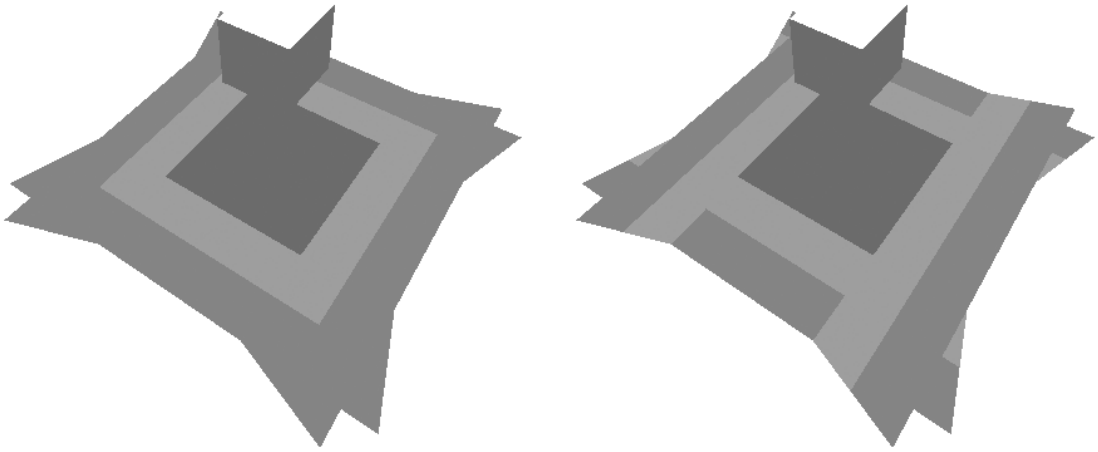
(a) (b)

Obrázek 5.4: Chyba vzniklá použitím hard shadow mapy pro určení kladného a záporného prostoru wedge. a) V rozích, kde se překrývají dvě wedge vznikne chyba vlivem špatného znaménka. b) Stejná chyba jako na obrázku (a), ale na malém stínu takřka nepatrná, lze pozorovat v levé dolní části.

praktické využití tohoto algoritmu je však podstatné. Jednou z možností je celý proces vytvářející stín pro jeden objekt opakovat pro každý objekt zvlášť a jednotlivé stínové mapy, které takto vzniknou, vhodně spojit. Při vykreslení více objektů, jejichž stíny se překrývají, přepíše později vykreslený objekt již hotový stín. Ukázku lze vidět na obrázku 5.6.

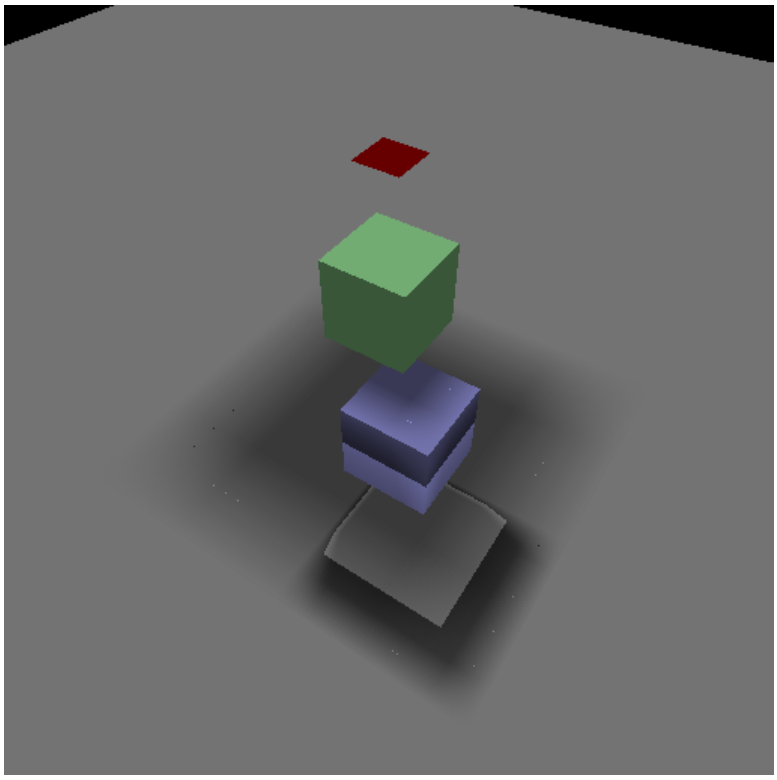
Další značnou nevýhodou je nevyhnutelná náročnost při vykreslování objektu s velkým množstvím polygonů. Sestavení wedge není nejjednodušší a ve velkém množství toto vytěžuje značně procesor. Pro další použití by bylo vhodné sestavení wedge přesunout z procesoru na grafické jádro, tedy do geometry shaderu.

Při vykreslení objektu s velkou hustotou polygonů též dochází ke vzniku artefaktů (5.7), které jsou z největší pravděpodobností způsobeny nedostatečnou kapacitou typu float. Při použití mnohem většího světla než je průměrná délka hrany siluety, tyto artefakty víceméně nahradí celý stín. Jednotlivé wedge tvořící penumbru se v takovéto situaci skoro celé ve velkém množství překrývají a je tedy třeba uchovávat příliš velké hodnoty ve stínové mapě.



(a) Prostory definovány hard shadow mapou. (b) Prostory definovány hard shadow quadem.

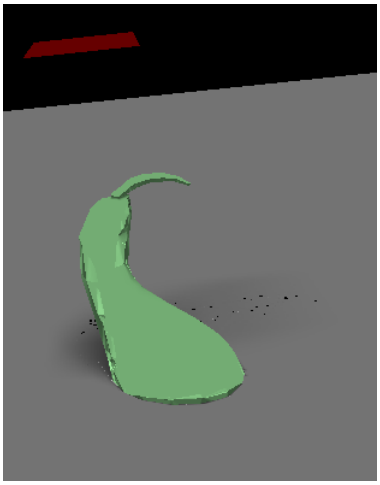
Obrázek 5.5: Rozdílně definované kladné a záporné prostory.



Obrázek 5.6: Krychle umístěná blíže ke světlu je vykreslována později, přepíše tedy stín krychle pod ní.

5.2 Stínové mapy

Variance Soft Shadow Mapping je v nedávné době uvedený algoritmus, který kombinuje několik starších technik a snaží se je spojit ve fungující celek. Pochopení tohoto algoritmu je obtížné a k jeho nastudování je třeba dohledat a nejlépe prakticky vyzkoušet vícero



Obrázek 5.7: Černé oblasti jsou artefakty vznikající při použití příliš velkého zdroje světla v poměru k průměrné délce hrany siluety.

dalších technik, na kterých je založen.

Největší komplikací při implementaci tohoto algoritmu je rozhodně nemožnost efektivně debuggovat shader programy pro OpenGL. Autoři článku o VSSM algoritmu, jak je nejspíše zvykem ve vědecké komunitě, vynechali spoustu praktických informací a zaměřili se jen na teorii. Naprogramovat tedy shader vytvářející stíny dle tohoto algoritmu je ze značné části experimentální záležitostí. Algoritmus popisuje své jednotlivé části, a když se nastudují a prakticky vyzkouší i algoritmy, ze kterých je VSSM odvozeno, lze naprogramovat jednotlivé části VSSM, avšak jejich spojení stále zůstává částečnou záhadou a vyžaduje řadu experimentů.

Výsledkem mé práce na VSSM jsou tedy bohužel jen jednotlivé díly pro sestavení tohoto algoritmu, ale ani po týdnech snahy o jejich vzájemnou spolupráci se mi nepodařilo naprogramovat výsledný shader tak, aby zobrazoval něco jiného než černý obraz.

Hlavním důvodem, proč se mi úspěšně nepodařilo dokončit implementaci VSSM algoritmu, je absence definice vstupu a výstupu jednotlivých částí VSSM. Kupříkladu není vůbec specifikováno jak má vypadat variance shadow map. VSSM jen říká, že se použije, ale nespecifikuje, jestli se má použít mapa momentů, nebo až zpracovaná mapa, tedy výsledky Čebyševovy nerovnosti. S největší pravděpodobností platí, že se mají vzít momenty, ale absence této informace v algoritmu dosti komplikuje jak jeho pochopení, tak jeho praktickou aplikaci.

Pro plynulý a kvalitní vjem z pozorované scény je potřeba vygenerovat alespoň 30 snímků za vteřinu, respektivě 60 snímků. Render jednoho snímku tedy nesmí zabrat více než 33,3ms, resp. 16,67ms. Generování Summed Area Table a Hierarchic Shadow Map může být časově docela náročné. Odzkoušel jsem tedy moji implementaci těchto technik.

SAT jsem naimplementoval pomocí článku [13]. Vygenerování takto naimplementované tabulky trvá maximálně jednu milisekundu a to i pro rozlišení vstupní textury 2048x2048. K implementaci HSM jsem nepoužil žadný algoritmus, avšak výsledkem mé vlastní práce je generování HSM za zhruba stejné časy jako SAT.

Pro studijní účely a jako základní stavební kámen VSSM jsem též naimplementoval základní VSM algoritmus.

Kapitola 6

Závěr

Metody pro zpracování měkkých stínů jsou rozsáhlou oblastí počítačové grafiky, do které neustále přibývají nové poznatky. V této práci jsem popsal dva odlišné přístupy ke tvorbě měkkých stínů. Nejdříve jsem popsal algoritmus stínových těles. Hlavní výhodou algoritmů stínových těles je jejich přesnost a ani algoritmus, který jsem zvolil není v této oblasti vyjímkou. Ačkoliv, díky principu jakým probíhá rasterizace jednotlivých částí stínu, uvadí jisté chyby, které na jeho výsledné kvalitě v určitých případech značně ubírají.

Základem algoritmu stínových těles jsou tvrdé stíny, získané pomocí obdélníků generovaných na základě siluety objektu vůči světlu. Takto získané tvrdé stíny jsou pak upraveny na měkké tak, že se sestaví geometrie polostínu (wedge), která se rasterizuje pomocí fragment shader programu. Algoritmus stínových těles jsem úspěšně naimplementoval a vyzkoušel různé přístupy, jak řešit některé jeho části. V kapitole o výsledcích jsou detailně popsány jednotlivé problémy, které tento algoritmus přináší, proč vznikají a způsoby jejich řešení.

Dalším algoritmem, kterému se tato práce věnovala, je Variance Soft Shadow Mapping. K tomuto algoritmu je uvedeno značné množství teorie, obsahující převážně matematické podklady pro tento algoritmus. Pro VSSM algoritmus je též uvedena další teorie k hlavním algoritmům, ze kterých vychází. Implementace této úpravy stínových map bohužel nedosáhla zdárného konce. Jednotlivé části popsané tímto algoritmem se podařilo úspěšně naprogramovat, ale nepodařilo se tyto části spojit do fungujícího celku, převážně z důvodu absence přesné informace o vstupech a výstupech jednotlivých částí v samotném popisu algoritmu.

Po teoriích k jednotlivým algoritmům je v této práci detailně popsána implementace jednotlivých algoritmů. Popis obsahuje nejdůležitější části implementace. Části, které bylo třeba domyslet, jelikož v daném algoritmu jsou pouze naznačeny, nebo části, které jsou nějakým způsoben problematické či jinak zajímavé.

Bohužel díky nekompletní implementaci druhého algoritmu nebylo možné provést porovnání výsledné kvality a hardwarové náročnosti obou přístupů. Lze se pouze referovat na původní články o těchto algoritmech, ze kterých vyplývá, že VSSM je dostatečně rychlé pro realtime aplikace a vzhledově dostatečně aproximuje přesnost s jakou vytváří stíny algoritmy stínových těles.

Literatura

[1] Akenine-Möller, T.; Assarsson, U.: Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. *13th Eurographics Workshop on Rendering*, 2002: s. 309–318.

[2] Annen, T.; Dong, Z.; Mertens, T.; aj.: Real-time, all-frequency shadows in dynamic scenes. In *ACM Trans. Graph papers*, ročník 27, 2008.

[3] Annen, T.; Mertens, T.; Seidel, H.-P.: Convolution Shadow Maps. In *Eurographics Symposium on Rendering (2007)*, ročník 18, 2007, s. 51–60.

[4] Assarsson, U.; Akenine-Möller, T.: A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Trans. Graph.*, ročník 22, č. 3, Červenec 2003: s. 511–520, ISSN 0730-0301, doi:10.1145/882262.882300.

[5] Bunnel, M.: Dynamic ambient occlusion and lightning. GPU Gems 2, Addison Wesley, 2005.

[6] Bunnell, M.; Pellacini, F.: Chapter 11. Shadow Map Antialiasing. GPU Gems, 1997.

[7] Crow, F. C.: *Shadow Algorithms for Computer Graphics*. ACM, 1977.

[8] Crow, F. C.: Summed-Area Tables for Texture Mapping. *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ročník 18, 1984: s. 207–212.

[9] Décoret, X.: N-buffers for efficient depth map query. Computer Graphics Forum 24, 3, 2005.

[10] Donnelly, W.; Lauritzen, A.: Variance shadow maps. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games, ACM*, Press, 2006, s. 161–165.

[11] Drettakis, G.; Fiume, E.: Shadow Algorithms for Computer Graphics. 1994.

[12] Fernando, R.: Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, 2005, str. 35.

[13] Hensley, J.; Scheuermann, T.; Coombe, G.; aj.: Fast summed-area table generation and its applications. *Computer Graphics Forum*, ročník 24, 2005: s. 547–555.

[14] Kaplan, W.: *Advanced Calculus*, kapitola §5.5 Green's theroem. Addison Wesley Publishing Company, Čtvrté vydání, 1991, s. 286–291.

[15] Markosian, L.; Kowalski, M. A.; Trychin, S. J.; aj.: Real-Time Nonphotorealistic Rendering. In *SIGGRAPH 97 Conference Proceedings*, 1997, s. 415–420.

[16] Yang, B.; Dong, Z.; Feng, J.; aj.: Variance Soft Shadow Mapping. ročník 29, č. 7, 2010: s. 2127–2134.